**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.

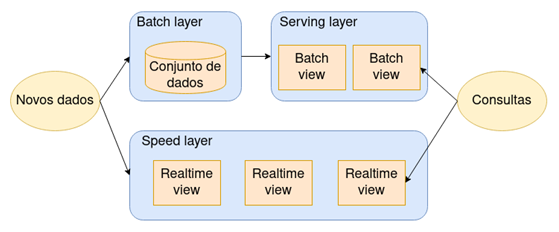
**TEMA 1 – PROCESSAMENTO DE FLUXOS DE DADOS DISTRIBUÍDOS**

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.

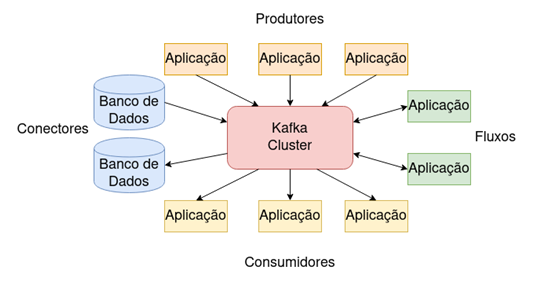
**2.4 ARQUITETURA**

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;

<liA 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.</li

Figura 2 – Arquitetura Kafka



Fonte: Kafka Apache, S.d.

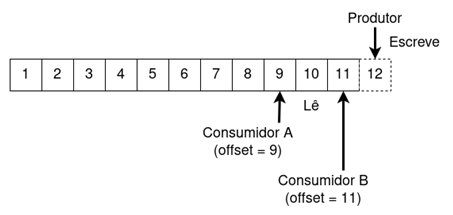
A comunicação em Kafka entre os clientes e os servidores é realizada utilizando-se o protocolo TCP. Dessa forma, a comunicação ocorre de forma simples, com alta performance e sem dependência com nenhuma linguagem de programação. O Kafka oferece um cliente Java, mas há clientes em diversas outras linguagens.

**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*;

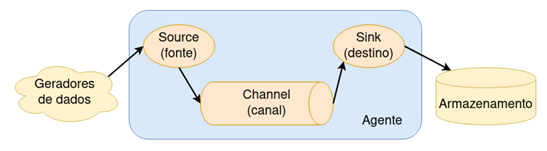
 ***Runners*:** é um componente interno às *sources* e aos *sinks* responsável por sua execução. Normalmente o usuário não precisa tomar conhecimento de sua execução, mas pode ser útil para os desenvolvedores de *sources* e *sinks* entenderem os detalhes de seu funcionamento;

 **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.

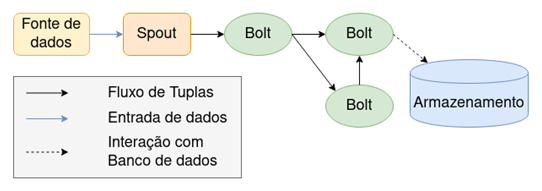
**4.1 TOPOLOGIA E DEMAIS CONCEITOS**

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;

<li**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.</li**

**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:

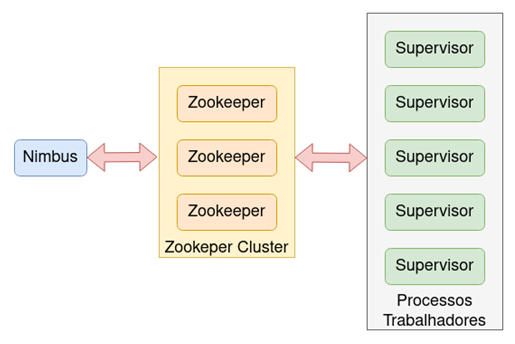
* ***Shuffle grouping*:** *agrupamento embaralhado*, ou seja, tuplas são distribuídas aleatoriamente às tarefas dos *bolts* de forma que as tuplas sejam distribuídas de maneira igualitária;
* ***Fields grouping*:** *agrupamento por campo*, ou seja, as tuplas são distribuídas de acordo com determinado 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.

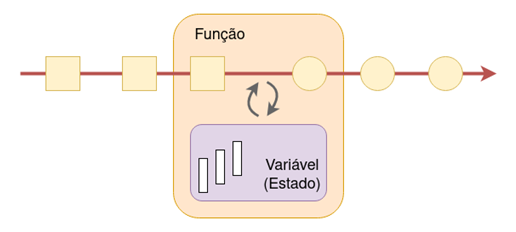
O Flink é projetado para executar aplicações de fluxos de eventos com estados em qualquer escala. Aplicações são paralelizadas em milhares de tarefas que são distribuídas e executadas concorrentemente em um *cluster*. Portanto, uma aplicação pode escalar horizontalmente de forma virtualmente ilimitada. 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.

**BIG DATA**

AULA 5

Prof. Luis Henrique Alves Lourenço

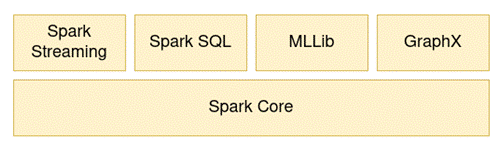
**CONVERSA INICIAL**

Nesta aula, será apresentada a você uma explicação dos principais conceitos e componentes do sistema de computação em *cluster* Spark. Primeiramente serão apresentados os conceitos necessários para entender como ocorre o processamento nos módulos mais básicos do Spark. Em seguida, vamos explorar as principais ferramentas adicionais do Spark: Spark SQL, MLLib, Spark *streaming* e GraphX.

**TEMA 1 – SPARK**

Spark é um sistema de computação em *cluster* para propósito geral criado e mantido pela Fundação Apache. É uma das ferramentas de processamento de *big data* mais interessantes atualmente. Muitas empresas utilizam Spark para solucionar problemas reais e realizar análises em conjuntos de dados muito grandes. O Spark utiliza grafos acíclicos dirigidos (DAG) para otimizar o fluxo de trabalho, tornando-o muito eficiente. Além disso, Spark possui um rico ecossistema de bibliotecas nativas que implementam APIs em alto nível para diversos usos distintos.

Figura 1 – Componentes do Spark

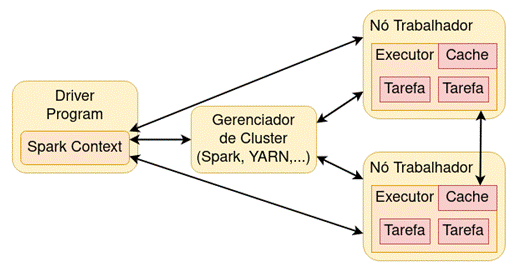


Como podemos ver na Figura 1, os componentes nativos do Spark são Spark *streaming*, Spark SQL, MLLib e GraphX. O Spark *streaming* foi projetado para realizar a ingestão e o processamento de dados em tempo real. O Spark SQL implementa uma interface SQL para o Spark de forma a permitir a utilização de consultas SQL e o processamento de dados estruturados no Spark. A MLLib é a biblioteca de aprendizado de máquina (*machine learning*) do Spark para que seja possível extrair significado do conjunto de dados. O GraphX implementa funções relativas à teoria dos grafos em Spark.

**1.1 ARQUITETURA DE *CLUSTER* SPARK**

Com Spark é possível dividir e distribuir o processamento de conjuntos de dados imensos em *clusters*. Aplicações Spark, ou*driver programs*executam como conjuntos de processos independentes, conhecidos como processos executores, em um *cluster* coordenados por um *SparkContext* e devem receber e aceitar conexões de seus processos executores durante a sua execução. Para executar em um *cluster*, o *SparkContext* utiliza um gerenciador de *cluster* para alocar recursos entre as aplicações. Dessa forma, é possível obter escalabilidade horizontal. O *cluster* Spark pode ser gerenciado utilizando o Hadoop YARN, o Apache Mesos, Kubernetes, ou o gerenciador de *clusters* nativo do Spark. Ou seja, o Spark pode atuar com diversos gerenciadores de *cluster*, contanto que eles possam instanciar processos executores e que tais processos possam se comunicar entre si.

Figura 2 – Arquitetura Spark



Fonte: Cluster..., S.d.

O gerenciador de *cluster* divide o trabalho de cada aplicação em vários processos executores, que se mantêm ativos durante toda a execução da aplicação executando tarefas em *threads*. Dessa forma, é possível isolar diferentes aplicações entre si, tanto do lado do escalonador de tarefas (no *driver program*) quanto no lado dos executores de tarefas, em que as tarefas de diferentes aplicações executam em diferentes JVMs. Isso significa que dados de diferentes aplicações não podem ser compartilhadas senão por meio de sua escrita em um sistema de armazenamento externo. Uma vez que a aplicação escalona tarefas para o *cluster*, ela deve executar fisicamente próxima aos nós trabalhadores, preferencialmente na mesma rede, para obter melhor desempenho.

**1.2 RDD E SPARKCONTEXT**

Podemos dizer que cada aplicação Spark consiste de um *driver program* que executa a função *main* e coordena a execução de várias operações paralelas no *cluster*. O Spark é projetado ao redor de um conceito central: o *Resilient Distributed Dataset* (RDD). RDDs são conjuntos de dados distribuídos e resilientes, ou seja, consistem em uma coleção de elementos particionados pelo *cluster* que podem ser operados em paralelos. Existem duas formas de criar um RDD: paralelizando uma coleção de dados existente no driver program ou referenciando um conjunto de dados em um sistema de armazenamento externo tal como um sistema de arquivos compartilhados, HDFS, HBase, ou qualquer fonte de dados que ofereça um *Hadoop InputFormat*. RDDs podem ser configurados pelo usuário para serem mantidos de forma persistente em memória, permitindo que sejam reutilizados de forma eficiente entre operações paralelas. Além disso, RDDs possuem a característica de serem tolerantes a falhas, ou seja, podem ser recuperados automaticamente em nós falhos.

A primeira coisa que um programa Spark faz é criar um objeto *SparkContext* que indica como o Spark deve acessar o *cluster*. O*SparkContext* é o componente responsável por tornar os RDDs resilientes e distribuídos.Para criar um *SparkContext* é necessário primeiro definir um objeto *SparkConf* que armazena a configuração de sua aplicação. Isso pode ser realizado em Python da seguinte forma:

Figura 3 – Definição do objeto *SparkConf*



Os RDDs são criados por um *SparkContext* no Driver Program por meio da aplicação do método *paralelize* sobre um conjunto de dados da seguinte forma:

Uma vez criado, o RDD pode ser executado em paralelo. Um parâmetro importante para coleções paralelas é a quantidade de partições que serão criadas com base num conjunto de dados. O Spark irá executar uma tarefa para cada partição no *cluster*. O ideal é que haja de 2 a 4 partições para cada CPU no *cluster*. Normalmente o Spark tenta definir o número de partições automaticamente baseado no *cluster*. Entretanto, é possível definir o número de partições manualmente passando a quantidade desejada como o segundo parâmetro do método *paralelize* (por exemplo: *sc.paralelize(data,10)*).

Figura 4 – Definição do número de partições

data = [1, 2, 3, 4, 5]
distData = sc.parallelize(data)


O Spark também é capaz de criar RDDs baseado em qualquer fonte de armazenamento suportado pelo Hadoop, incluindo o HDFS, Cassandra, HBase, Amazon S3, dentre outros. Além disso, o Spark também possui suporte a arquivos locais formatados em JSON, CV, arquivos sequenciais, arquivos de objeto e vários outros. Nesse caso, RDDs são criados por meio do método *textFile* do *SparkContext*. Esse método recebe uma URI para o arquivo que é lido como uma coleção de linhas. A URI pode ser tanto o caminho para um arquivo local, como um endereço. A leitura de um arquivo em um sistema HDFS pode ser feita da seguinte forma:

Figura 5 – Leitura de um arquivo em um sistema HFDS



Outra abstração importante em Spark são as variáveis compartilhadas que podem ser utilizadas em operações paralelas. Por padrão, quando o Spark executa uma operação em paralelo como um conjunto de tarefas em nós diferentes, ele realiza uma cópia de cada variável utilizada pela função em cada tarefa. Em alguns casos, uma variável precisa ser compartilhada por diversas tarefas ou entre tarefas e o *driver program*. Dessa forma, o Spark suporta dois tipos de variáveis compartilhadas: *variáveis de broadcast*, que podem ser utilizadas como *cache* para armazenar um valor em memória em todos os nós; e os *acumuladores* que são variáveis que apenas acrescentam valores, como contadores ou somadores.

**1.3 OPERAÇÕES EM RDDS**

RDDs suportam basicamente dois tipos de operações: *transformações*, que criam um novo conjunto de dados de outro já existente, e *ações*, que retornam um valor ao *driver program* após realizar a computação de um conjunto de dados. Por exemplo, *map* é uma transformação que passa cada elemento de um conjunto de dados por uma função e retorna um novo RDD representando os resultados, ao passo que *reduce* é uma ação que agrega todos os elementos de um RDD utilizando alguma função e retorna o resultado final ao *driver program.*

Todas as transformações em Spark são preguiçosas, ou seja, não produzem resultados imediatamente. Em vez disso, elas apenas memorizam as transformações aplicadas a algum conjunto de dados. As transformações são computadas apenas quando uma ação requisita um resultado para ser retornado para o Driver Program. Dessa forma, o Spark é capaz de calcular a forma mais eficiente de executar a computação dos dados.

Por padrão, cada RDD transformado pode ser recomputado a cada vez que uma ação é executada sobre ele. No entanto, é possível tornar um RDD persistente em memória utilizando o método *persist* (ou *cache*), no qual o Spark irá manter os elementos armazenados no *cluster* para otimizar acessos futuros. Também é possível realizar a persistência de RDDs em disco, ou replicado por meio de diversos nós.

Uma das coisas mais difíceis em Spark é entender o escopo e o ciclo de vida das variáveis e métodos executando pelo *cluster*. Operações RDD que modificam variáveis fora de seu escopo podem ser uma grande fonte de confusões. Para evitar essas confusões, é sempre importante lembrar que os elementos de um conjunto de dados executarão de forma paralela. Dessa forma, é impossível garantir que existirá uma ordenação entre as execuções de cada tarefa.

Segundo a documentação oficial do Spark o conjunto de transformações incluem: *map*, *filter*, *flatMap*, *mapPartitions*, *mapPartitionsWithIndex*, *sample*, *union*, *intersection*, *distinct*, *groupByKey*, *reduceByKey*, *agregateByKey*, *sortByKey*, *join*, *cogroup*, *cartesian*, *pipe*, *coalesce*, *repartition*, *repartitionAndSortWithinPartitions*, entre outras.

O conjunto de ações inclui *reduce*, *collect*, *count*, *first*, *take*, *takeSample*, *takeOrdered*, *saveAsTextFile*, *saveAsSequenceFile*, *saveAsObjectFile*, *countByKey*, *foreach*, entre outras.

Certas operações no Spark desencadeiam um evento conhecido como *shuffe* (ou embaralhamento). O *shuffle* é um mecanismo em Spark utilizado para redistribuir os dados de forma a serem agrupados de uma forma diferente pelas partições. Normalmente esse tipo de operação envolve cópias de dados entre executores e nós, uma vez que exige muitas leituras em disco, serialização de dados, e tráfego de rede. Portanto trata-se de uma operação cara e complexa.

**TEMA 2 – SPARK SQL**

Spark SQL é o módulo do Spark utilizado o processamento de dados estruturados. Diferentemente da API básica de RDDs do Spark, a interface fornecida pelo Spark SQL oferece mais informações sobre a estrutura tanto dos dados quanto da computação a ser realizada. Internamente essa informação extra é utilizada para otimizações adicionais.

Um dos usos do Spark SQL é poder realizar consultas SQL sobre os dados. No entanto, também é possível utilizá-lo para consultar dados em uma instalação do Hive. Para isso é necessário instanciar uma *SparkSession* com suporte ao Hive, incluindo a configuração necessária para conectar a um *metastore* Hive permanente, suporte a serializadores e desserializadores Hive e funções Hive definidas por usuários. O suporte ao Hive pode ser habilitado mesmo quando não for necessário criar uma base de dados Hive nova, inclusive criando toda a configuração necessária para operar.

Porém, a fonte de dados padrão utilizada pelo Spark para todas as operações são arquivos *parquet*, a não ser que sejam configurados de outra forma. É possível especificar manualmente qual o formato da fonte de dados que será utilizado. Os formatos de dados das fontes são especificados como um parâmetro extra. Em Python os dados podem ser carregados e escritos da seguinte forma:

Figura 6 – Escrita de dados em python

df = spark.read.load("examples/src/main/resources/people.json", format="json")
df.select("name", "age").write.save("namesAndAges.parquet", format="parquet")


Os formatos de fontes de dados suportados de maneira nativa pelo Spark SQL são:

* ***Parquet*:** formato de armazenamento colunar suportado por diversos sistemas e disponível em qualquer projeto do ecossistema Hadoop independente da escolha do *framework* de processamento de dados, modelo de dados ou linguagem de programação. É um formato projetado para suportar esquemas de compressão e codificação muito eficientes com base em estruturas aninhadas complexas;
* ***JSON*:** *Javascript object notation*. É um formato leve projetado para troca de dados e que possa ser de fácil leitura e escrita por seres humanos. Baseia-se em uma coleção de pares chave-valor;
* ***ORC*:** *Optimized row columnar*. É um formato de arquivo que permite o armazenamento altamente eficiente de arquivos Hive. Projetado para superar limitações de outros formatos de arquivos Hive;
* ***CSV*:** *Comma-separated values.* É um formato de arquivo que armazena texto de forma tabular que utiliza vírgulas para separar valores. Cada linha do arquivo é um registro. Cada registro é constituído de um ou mais campos, separados por vírgulas;
* ***LibSVM*:** é a implementação da API de fonte de dados do Spark para *Support-vector* *machines*, que são modelos de aprendizagem supervisionada associados a algoritmos de aprendizados que analisam dados utilizados em clarificação e análise de regressão;
* ***JDBC*:** é a fonte de dados utilizada para ler dados em outros bancos de dados. As tabelas de dados externos podem ser lidas como *dataframes* ou *views* temporárias do Spark SQL. Permite ao Spark SQL atuar como um motor de consultas distribuídas utilizando conectores JDBC/ODBC ou também por meio da linha de comando.

Além disso, o Spark SQL também suporta fontes de dados em arquivos texto. Para fontes de dados baseadas em arquivos, como texto, *parquet* ou *json*, é possível especificar um caminho onde a tabela será criada. Quando a tabela é removida, o caminho é preservado assim como os dados da tabela, porém, se nenhum caminho for especificado, o Spark SQL armazenará os dados em uma tabela padrão sobre o diretório *warehouse*. Nesse caso, quando a tabela é removida, o caminho da tabela é juntamente removido.

**2.1 DATAFRAMES E DATASETS**

Um *dataset* é uma coleção de dados distribuída implementada como uma interface no Spark 1.6, que oferece os mesmos benefícios que um RDD (tipagem forte e a habilidade de utilizar funções lambda, que são a maneira de utilizar uma função sem a necessidade de defini-la) em conjunto com os benefícios do motor de execução otimizado do Spark SQL. Com base no Spark 2.0, o uso de *Datasets* é priorizado sobre os *dataframes*. *Datasets* podem ser criados com base em objetos JVM e então manipulados por meio de transformações funcionais como *map*, *flatmap*, *filter*, entre outros. A API *dataset* está disponível em Java ou Scala, porém não há uma API *dataset* implementada em Python ou R. No entanto, devido à natureza dinâmica dessas linguagens, muitos dos benefícios da API *Dataset* já estão disponíveis.

Um *dataframe* é um *dataset* estruturado em colunas nomeadas. Conceitualmente é equivalente a uma tabela de um banco de dados relacional ou um *dataframe* em Python ou R, porém com melhores otimizações. De outra forma, é possível definir *dataframe* como sendo uma extensão dos conceitos de RDD e *dataset* ao adicionar uma forma de estruturar seus dados. *dataframes* podem ser criados com base em diferentes fontes de dados tais como: arquivos de dados estruturados, tabelas Hive, bases de dados externos, ou RDDs. A API *dataframe* está disponível em Java, Scala, Python e R. Uma vez que os dados em um *dataframe* são estruturados em linhas e colunas, ou seja, possuem um esquema (*schema*), é possível realizar consultas SQL. Além disso, a adoção de um esquema permite ao Spark realizar operações de forma ainda mais otimizada. Dessa forma, *dataframes* e *datasets* têm sido adotados em todos os módulos Spark como forma de otimizar a manipulação dos dados.

**2.2 SUBMETENDO APLICAÇÕES**

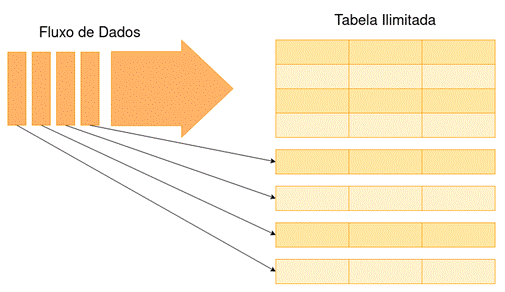
Para submeter aplicações ao *cluster*, há um *script* no diretório *bin* do Spark chamado *spark-submit*, que pode ser utilizado para submeter aplicações em qualquer os gerenciadores de *cluster* suportados pelo Spark. Se a aplicação a ser submetida ao *cluster* depende de outros projetos, deve-se empacotá-las junto com a aplicação para que sejam distribuídas pelo *cluster*. Para aplicações Java ou Scala, um *assembly jar* (ou *uber jar*) deve ser criado contendo o código da aplicação e suas dependências, já para aplicações Python pode-se utilizar o argumento --*py-files* do *spark-submit* para adicionar arquivos *.py,* *.zip* ou *.egg*. Para o caso em que a aplicação depende de muitos arquivos Python, é recomendado que sejam empacotados em arquivos *.zip* ou *.egg*.

**2.3 STRUCTURED STREAMING**

*Structured streaming* é um motor de processamento de fluxos escalável e tolerante a falhas criado com base em Spark SQL. Ele permite a definição de computações em fluxo da mesma forma que seriam definidos no processamento em dados estáticos. O motor Spark SQL mantém de forma contínua e incremental a atualização dos resultados finais à medida que os dados continuam a ser recebidos. Para isso, desde Spark 2.0, as APIs *dataset* e *Dataframe* podem ser utilizadas em conjunto com as linguagens Java, Scala, Python ou R para representar não apenas dados estáticos e limitados, mas também dados ilimitados como fluxos.

Internamente, consultas *Structured* *streaming* são processadas utilizando um motor de processamento de lotes muito pequenos. Dessa forma, os fluxos de processamento são processados como uma série de pequenos trabalhos em lote de forma que seja possível alcançar latências fim a fim tão baixas quanto 100 milissegundos com garantias de tolerância a falhas. Como podemos ver na Figura 7, é como se os dados do fluxo fossem representados em uma tabela ilimitada e cada nova entrada é processada de forma incremental em que uma consulta no fluxo de dados gera um resultado em tabela que é atualizada a cada intervalo onde novos dados são recebidos.

Figura 7 – Fluxos de dados como tabelas ilimitadas



Fonte: Spark SQL, S.d.

Desde o Spark 2.3, um novo modo de processamento de baixa latência foi introduzido com o nome de *processamento* *contínuo* que pode atingir latências próximas a 1 milissegundo. Não é necessário realizar modificações às operações *dataset* e *dataframe* de nenhuma consulta para alterar o modo de processamento. Para isso, basta alterar os requisitos da aplicação.

**TEMA 3 – SPARK *STREAMING***

Spark *streaming* é uma extensão da API *core* do Spark que permite o processamento de fluxo de dados de forma escalável, com alta eficiência e tolerante a falhas. Os dados podem ser importados de diversas fontes tais quais Kafka, Flume, Kinesis, sockets TCP, além de sistemas de armazenamento distribuído como HDFS e S3. Com isso, os dados podem ser processados por algoritmos complexos         expressados em funções de alto nível como *map*, *reduce*, *join,* *window*, até algoritmos de *machine learning*, processamento em grafos ou, como vimos no tema anterior de nossa aula, consultas SQL. E, finalmente, os dados processados podem ser destinados a sistemas de armazenamento, bancos de dados e *dashboards*.

Figura 8 – Spark *streaming*

Fonte: Spark Apache, S.d.

Internamente, o Spark *streaming*, como ilustrado pela Figura 6, recebe um fluxo de dados que é dividido em pequenos lotes antes de ser processado pelo *core* Spark para gerar o fluxo final de resultados em lote (*batch*). Ou seja, o fluxo de dados recebido é discretizado em pequenas partes para determinado intervalo de tempo resultando em um único RDD. Cada RDD no Spark *streaming*, então, representa o conjunto de dados recebidos em um dado intervalo de tempo. Dessa forma, cada RDD pode ser processado da mesma forma que qualquer outro RDD pelo cluster Spark. É importante ressaltar que, pela forma como o Spark processa o fluxo de dados de forma discretizada em RDDs, embora o processamento dos dados possa ser realizado com uma latência muito baixa, não pode ser considerado de fato em tempo-real. Esse tipo de processamento é conhecido como processamento de *microbatches*, ou seja, é a prática de coletar dados em pequenos grupos (*batches*) para o propósito de realizar alguma ação sobre os dados. Portanto, o processamento do Spark *streaming* não ocorre em tempo real, porém na prática ele opera de forma muito próxima a isso. Também é valido lembrar que o processamento do Spark pode ser realizado de forma paralela em um *cluster*.

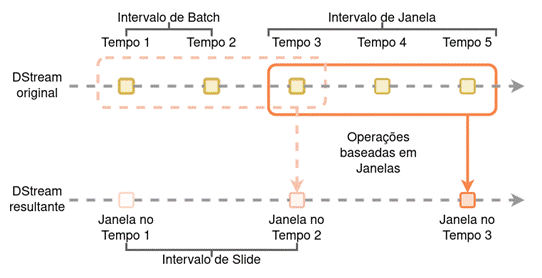
**3.1 FLUXOS DISCRETIZADOS (*DSTREAMS*)**

O Spark *streaming* oferece uma abstração de alto nível conhecida como *DStreams* (ou *discretized streams*) que representa um fluxo contínuo de dados. *DStreams* podem ser criados tanto para receber dados de fontes tais como Kafka, Flume e Kinesis, ou para aplicar operações de alto nível em outros DStreams. Internamente, um *DStream* é representado como uma sequência de RDDs. É possível realizar transformações em um *DStream* quase como em um RDD, tais como *map*, *flatMap*, *filter*, *reduceByKey*, entre outras. Além disso, também é possível manter o estado dos dados em um *DStream*. Isso é muito útil para realizar operações que dependem de informações entre um RDDs.

**3.2 TRANSFORMAÇÕES DE JANELA**

O Spark *streaming* oferece a capacidade de realizar transformações em janelas, o que permite aplicar transformações em um conjunto de RDDs para computar os resultados de um intervalo maior. Ou seja, transformações aplicadas a uma janela são aquelas que acumulam RDDs em um intervalo de tempo maior e os processa de forma conjunta, gerando um resultado acumulado. Com o passar do tempo, a janela se desloca incluindo novos RDDs, eliminando os que se encontram além do intervalo da janela.

Figura 9 – Transformações de janelas

  
Fonte: Spark Apache, S.d.

É importante destacar a diferença entre alguns intervalos de tempo que precisamos definir para que os resultados reflitam a informação que queremos extrair. O intervalo de *batch* é a quantidade de tempo que define a frequência em que os dados são capturados do fluxo pelo *DStream*, ou seja, a quantidade de tempo entre cada captura de dados. O intervalo de *slide*, de forma semelhante ao intervalo de *batch, é* a quantidade de tempo que define a frequência em que as transformações são aplicadas à janela. E o intervalo de janela, ou comprimento da janela, é a duração da janela. Por fim, devemos destacar que tanto o intervalo de *slide* quanto o intervalo de janela devem ser múltiplos do intervalo de *batch*.

Na Figura 9, temos a demonstração de como funciona o processamento de transformações em janelas. Na linha pontilhada superior (*DStream* original), temos os RDDs da *DStream* que recebeu os dados do fluxo. Um RDD é gerado a cada uma unidade de tempo. Circulando os RDDs, temos a representação de uma janela. Nesse exemplo, cada janela possui uma duração (ou intervalo de janela) de 3 unidades de tempo. Ou seja, ela contém os dados gerados em 3 unidades de tempo. E na linha pontilhada inferior (*DStream* resultante) temos o *DStream* que recebe os resultados da transformação gerada pelas janelas da l *DStream* original. Seu intervalo de *slice* é de 2 unidades de tempo. Portanto, nesse exemplo, a cada 1 unidade de tempo um RDD é gerado pela *DStream* original, e a cada 2 unidades de tempo, uma janela transforma os dados de 3 unidades de tempo atrás da *DStream* original e os armazena na *DStream* resultante. Essa transformação pode ser exemplificada por uma linha de código em Python como a seguinte, que é utilizada para realizar a contagem de palavras em um texto:

Figura 10 – Linha de código em Python

# Aplica o Reduce aos dados capturados nos últimos 30 segundos,
# a cada 10 segundos
windowedWordCounts = pairs.reduceByKeyAndWindow(lambda x, y: x + y, lambda x, y: x - y, 30, 10)


**3.3 STRUCTURED STREAMING**

Como vimos quando abordamos o Spark SQL, há um novo motor de processamento de fluxos baseado no Spark SQL que oferece uma API mais nova que o *DStream* chamado de *structured streaming*. Internamente, o motor *structured streaming* realiza processamento de *microbatches* com as vantagens que o motor de processamento do Spark SQL oferece. Ou seja, é possível utilizar as APIs *dataset* e *dataframe* para realizar consultas em dados de fluxos de forma incremental e contínua. Com isso, o *structured streaming* permite tratar os dados como uma tabela que está constantemente crescendo. À medida que novos *batches* são incluídos para processamento, a tabela resultante é atualizada. No entanto, vale destacar que o *structured streaming* não materializa toda a tabela, mas lê apenas o último dado disponibilizado pela fonte de fluxo de dados, processa o dado de forma incremental para atualizar o resultado, e então descarta a fonte de dados. Para isso são armazenados apenas os estados dos dados que forem necessários para realizar o processamento e atualizar a tabela resultante.

**TEMA 4 – MLLIB**

MLLib é a biblioteca de aprendizado de máquina (*machine learning*) escalável e do Spark. O aprendizado de máquina é parte do campo da inteligência artificial que se refere ao estudo de modelos estatísticos para resolver problemas específicos com padrões e inferências. Esses modelos são treinados para resolver problemas específicos por meio do treinamento, utilizando dados criados com base no domínio do problema. Podemos dividir o aprendizado de máquina em três categorias de aprendizado: *supervisionado*, *não supervisionado* e *por reforço*:

 **Aprendizado supervisionado:** utiliza conjuntos de dados que contêm ambos os dados de entrada e os resultados esperados, ou seja, os dados de entrada são rotulados. Essa categoria pode ser ainda subdividida em dois conjuntos de algoritmos: *classificação* e *regressão*;

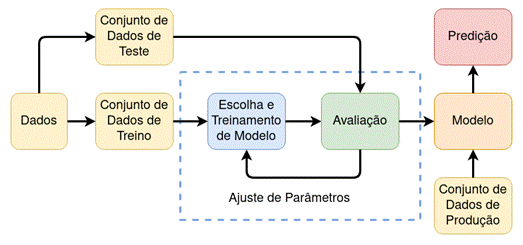
 **Aprendizado não supervisionado:** utiliza apenas conjuntos de dados de entrada e, portanto, podemos dizer que os dados não são rotulados previamente. Internamente funciona levando em conta a tentativa de identificar a estrutura inerente ao conjunto de dados. Algoritmos de *clustering* fazem parte desta categoria;

 **Aprendizagem por reforço:** baseado nos conceitos de reforço e punição emprestados da psicologia em que um agente tem por objetivo maximizar determinado parâmetro por meio da tomada ações. O agente é recompensado ou punido de acordo com as ações tomadas. A principal diferença para o aprendizado supervisionado é que os dados de entrada nunca são explicitamente relacionados com os seus rótulos.

**4.1 FLUXO DE APRENDIZADO DE MÁQUINA**

Podemos definir o funcionamento estrutural de um fluxo de aprendizado de máquina em algumas etapas, as quais etapas podem variar um pouco dependendo da especificidade de cada implementação, porém, de forma geral, todas possuem uma estrutura semelhante: Utilizamos um conjunto de dados de teste e de treino para escolher os modelos de aprendizado de máquina mais adequados, treiná-los e avaliá-los com o objetivo de alcançar um modelo que seja capaz de realizar predições úteis com base em dados do mundo real. Dessa forma, tal fluxo pode ser subdividido nas seguintes etapas:

Figura 11 – Fluxo de trabalho de aprendizado de máquina



* **Obtenção dos dados:** etapa em que os dados são obtidos. Os dados podem ser utilizados tanto para o treino dos modelos escolhidos quanto para os testes que avaliarão a eficiência dos modelos. Podem ser utilizados dados de diversas fontes, tais como arquivos, bancos de dados, sensores ou qualquer outra fonte suportada pelo Spark;
* **Preparação dos dados:** é uma das etapas mais importantes. Como vimos anteriormente, a maior parte do tempo gasto em análise de dados é utilizada para preparar os dados que serão utilizados para realizar a análise. Nessa etapa, podemos ter diversos tipos de dados com problemas: como os dados faltantes, dados com ruídos e dados inconsistentes, uma vez que durante o processo de produção e entrega dos dados podem ocorrer falhas técnicas ou humanas. Além disso, é nessa etapa que os dados são formatados da maneira mais adequada;
* **Escolha dos modelos adequados:** etapa em que os modelos mais eficientes para o tipo de dados são escolhidos para serem treinados;
* **Treinamento:** etapa em que o modelo utiliza um conjunto de dados de treino para aprender como processar a informação por meio de ajustes de parâmetros e tentativas de predição;
* **Avaliação:** etapa em que o modelo utiliza um conjunto de dados desconhecido, ou seja, o modelo não conhece o resultado que deve obter de tais dados. Por fim, o desempenho avaliado para ser utilizado em produção. Caso o desempenho não seja suficiente, o modelo passa por um novo ciclo de ajuste de parâmetros, treinamento e avaliação;
* **Ajuste de parâmetros:** uma vez que o modelo pode ser avaliado por um conjunto de dados que desconhece, o resultado da avaliação pode ser utilizado para ajustar seus parâmetros;
* **Predição:** etapa final que tem por objetivo responder uma questão utilizando dados reais. Também chamada de *inferência*, é onde se pode demonstrar o valor do aprendizado de máquina.

**4.2 COMPONENTES DA MLLIB**

Uma vez que conhecemos o funcionamento básico de um fluxo de aprendizado de máquina, somos capazes de entender os principais componentes da MLLib. As principais ferramentas das bibliotecas podem ser divididas nas seguintes categorias:

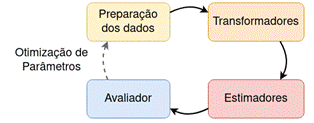
* **Algoritmos de aprendizado de máquina:** são os algoritmos mais comuns, como os de classificação, regressão, *clustering*, filtragem colaborativa, entre outros;
* **Caracterização:** é a categoria que engloba os algoritmos de extração de características, transformação, redução dimensionalidade e seleção;
* **Pipelines:** essa categoria inclui as ferramentas utilizadas na construção e otimização de *pipelines*;
* **Persistência:** é a categoria das ferramentas utilizadas para armazenar e carregar algoritmos, modelos e *pipelines*;
* **Utilidades:** inclui ferramentas de álgebra linear, estatística, manipulação de dados e outras ferramentas auxiliares.

A MLLib foi originalmente implementada com base em RDDs, porém essa API atualmente não está mais sendo desenvolvida e novas implementações não serão incluídas, e sua atualização se resume apenas à manutenção básica. Levando em conta o Spark 2.0, a API principal da MLLib é baseada na API *Dataframe* da Spark SQL. Essa mudança permitiu uma API muito mais simples e fácil, além de permitir também a entrada de dados de diversas fontes por meio dos *Datasources* que a API *Dataframe* implementa, a capacidade de realizar consultas SQL, novas otimizações e a unificação da API utilizando-se das diversas linguagens suportadas pela API *dataframe*. Dessa forma, podemos dizer que os objetivos da MLLib incluem as seguintes ações: simplificação do desenvolvimento e implantação de *pipelines* de aprendizado de máquina por meio das classes fornecidas pela biblioteca.

**4.3 *PIPELINES* DE APRENDIZADO DE MÁQUINA**

O componente mais importante da MLLib é o *pipeline*, pois é neste que se define o fluxo de trabalho da aplicação pelo encadeamento de uma série de algoritmos. Dessa forma, a MLLib padroniza uma API para algoritmos de aprendizado de máquina para tornar mais fácil a combinação de múltiplos algoritmos em um único pipeline. *Pipelines* em MLLib utilizam os tipos definidos pela Spark SQL para *dataframes*. Além disso, a API *dataframe* pode utilizar o tipo *vector* definida na API da MLLib. Os principais componentes de um *pipeline* são transformadores, estimadores e parâmetros.

Figura 12 – Conceitos básicos de um *pipeline*



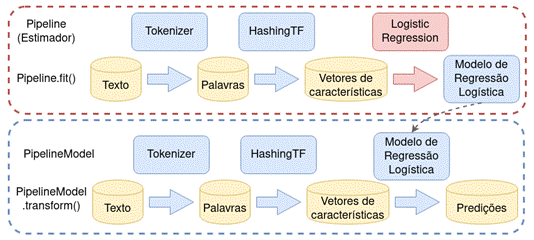
Um transformador é uma abstração que inclui transformadores de características ou conjunto de atributos de objetos utilizados para calcular a previsão e modelos aprendidos. É a etapa em que ocorre a extração de características e a transformação dos dados para o formato em que serão consumidas. Um *pipeline* pode possuir quantos transformadores encadeados forem necessários. Tecnicamente, um transformador implementa um método *transform()*, que converte um *dataframe* em outro, geralmente pela adição de novas colunas. Por exemplo, um transformador pode receber um *dataframe*, ler uma coluna, mapeá-la em outra coluna e retornar um novo *dataframe* com a coluna mapeada adicionada. Exemplos de transformadores: normalização, tokenização, conversão de valores categóricos, e outros.

Um estimador abstrai o conceito de um algoritmo de aprendizado, ou qualquer algoritmo de aprendizado que treina sobre um determinado conjunto de dados. É nessa etapa em que modelos são produzidos. O algoritmo de aprendizado treina utilizando os dados de treino e retorna um modelo. O modelo retornado atua como um transformador. Tecnicamente um estimador implementa um método *fit()* que recebe um *dataframe* e produz um modelo. Por exemplo, algoritmos como *LogisticRegression* é um estimador e chamar o método *fit()* treina um *LogisticRegressionModel*, que é um modelo e, portanto, um transformador.

Vale lembrar que os métodos *Transformador.transformer()* e *Estimador.fit()* não armazenam estados nas versões atuais da Spark MLLib. Dessa forma, para compartilhar parâmetros entre transformadores e estimadores utilizam uma API em comum.

Por fim, podemos especificar um *pipeline* como uma sequência de etapas ou estágios em que cada etapa é um transformador ou um estimador. As etapas de um *pipeline* devem ser efetuadas em uma ordem determinada. Os *dataframes* recebidos são transformados à medida que passam por cada estágio. Em cada estágio transformador, o método *transform()* é executado sobre os dados. E em cada estágio estimador o método *fit()* é chamado para produzir um transformador, que será parte do *PipelineModel*, cujo método *transform()* é utilizado para produzir um *dataframe*.

Figura 13 – Exemplo de *pipeline*

  
Fonte: ML..., S.d.

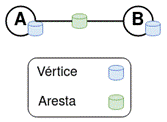
Na Figura 13, temos o exemplo de um pipeline de 3 estágios. Os dois primeiros (*Tokenizer* e *HashingTF*) são transformadores, e o terceiro (*LogisticRegression*) é um estimador. Os cilindros abaixo representam os dataframes. O método *Pipeline.fit()* é utilizado sobre o *dataframe* original que possui documentos de texto e rótulos (que são os resultados que o modelo deve prever para os dados de treinamento). O método *Tokenizer.transform()* divide o texto em palavras e, dessa forma adiciona uma coluna *palavras* ao *dataframe*. O método *HashingTF.transform()* converte a coluna de palavras em vetores de características (que são o conjunto de atributos de objetos utilizados para calcular a previsão) e adiciona uma nova coluna contendo os vetores ao *dataframe*. Logo após temos o estimador *LogisticRegression*. O *pipeline* executa *LogisticRegression.fit()* e produz um *LogisticRegressionModel*. Se o *pipeline* tivesse mais estimadores, o método *transform()* do *LogisticRegressionModel* seria necessário para gerar um *dataframe* para o próximo estimador. Tal *pipeline* funciona como um estimador, e portanto a execução do método *Pipeline.fit()* produz um *PipelineModel*, ou seja, um transformador. O *PipelineModel* é utilizado em testes, como podemos ver no segmento destacado pela linha pontilhada azul na Figura 13. Ali também podemos ver que o *PipelineModel* possui o mesmo número de etapas que o *pipeline* original, porém todos os estimadores foram substituídos por transformadores. Ao executar o método *PipelineModel.transform()* sobre o conjunto de dados, os dados passam por cada um dos transformadores do *PipelineModel*. Cada método *transform()* dos transformadores atualiza o conjunto de dados que são passados ao próximo estágio. Dessa forma, podemos garantir que os dados de treinamento e de teste passam pelo mesmo processamento de características.

**TEMA 5 – GRAPHX**

GraphX é o componente do Spark para computação de grafos em sistemas distribuídos de larga escala. Ele foi desenvolvido por meio de um projeto de pesquisa como forma de unificar o processamento de grafos e de sistemas paralelos até se tornar uma parte integral do projeto Spark. A API GraphX extende a abstração RDD em um multgrafo dirigido com propriedades em cada vértice e aresta para criar uma nova abstração conhecida por *Graph*. O suporte à computação em grafos é possibilizado por meio de um conjunto de operações fundamentais assim como um conjunto de algoritmos e construtores que simplificam tarefas analíticas utilizando grafos.

Segundo a teoria dos grafos, grafos são estruturas matemáticas semelhantes a um conjunto de objetos nos quais pares de objetos são relacionados de alguma forma. Esses objetos são representados utilizando vértices e suas relações são representadas por arestas, como podemos observar na Figura 14.

Figura 14 – Representação de grafos



Como já sabemos, técnicas de computação de grafos podem ser utilizadas para resolver diversos tipos de problemas de forma eficiente. Dessa forma, a computação paralela de grafos pretende auxiliar na tarefa de processar dados de grafos com um número imenso de arestas e vértices. Nesse caso, possivelmente os dados não caberiam no armazenamento de uma só máquina. Além disso, deve-se levar em consideração a complexidade de distribuir o processamento desse grafo em um *cluster*. Existem muitos algoritmos da teoria dos grafos que são amplamente utilizados em análise de dados e podem se beneficiar de tal abordagem, uma vez que muitos dados podem ser representados utilizando grafos de forma muito mais eficiente como é o exemplo de tabelas esparsas.  Com isso, é possível tornar a implementação de algoritmos que realizam computação em tais tipos de dados muito mais simples. Podemos citar como exemplos de algoritmos de grafos paralelos:

* ***PageRank*:** determina a importância de distintos nós em uma rede. Originalmente desenvolvido pelo Google para contabilizar a quantidade e a qualidade de referências (ou *links*) apontando para uma página e, dessa forma, permitir a classificação de páginas na internet;
* ***Triangle counting*:** mede a coesão de comunidades por meio da contagem de triângulos à qual cada nó de um grafo pertence. Um triângulo é um conjunto de três nós em que cada um deles possui uma relação com os outros dois. Dessa forma, podemos determinar que os nós que formam mais triângulos pertencem a um conjunto de nós, ou seja, uma comunidade e, portanto, são mais relacionados entre si. Além disso, esse algoritmo permite determinar o grau de estabilidade de um grafo. Esse tipo de algoritmo é útil para análise de redes sociais e para a computação de índices de rede tais como o coeficiente de *clustering*;

 ***Connected components*:** busca encontrar todos os componentes de um grafo que estão conectados entre si onde cada vértice (ou componente) dentro de um grupo pode ser alcançado com base em qualquer outro vértice do grupo. Além disso, não deve haver nenhum caminho, ou seja, arestas, entre dois grupos distintos.

**5.1 API GRAPHX**

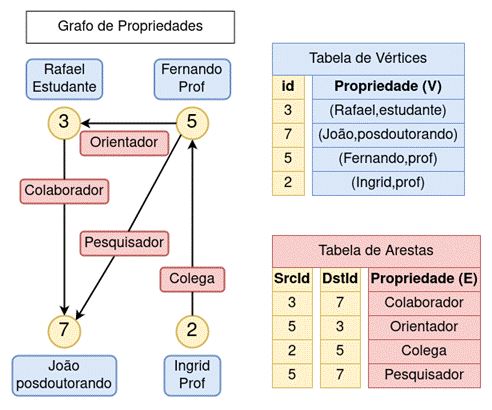
A API GraphX oferece uma forma de armazenar grafos na forma de tabelas e utilizar operações de tabelas para expressar operações de grafos. Assim, facilitando o armazenamento de dados que são mais bem expressos na forma de grafos, e permitindo operar sobre eles da mesma forma como é realizado com qualquer abstração de dados no ecossistema Spark. Para isso, o GraphX modela grafos utilizando uma abstração conhecida como grafos de propriedades (*Property Graphs*). Um grafo de propriedades é um multigrafo dirigido com objetos (ou propriedades) definidos por usuário em cada vértice e aresta. Um grafo dirigido é um grafo cujos vértices possuem uma direção, ou seja, a ordem das arestas importa. Portanto, podemos definir um multigrafo como um grafo dirigido em que múltiplos vértices paralelos podem compartilhar o mesmo par de arestas (fonte e destino). Da mesma forma que RDDs, grafos de propriedades são imutáveis, distribuídos e tolerantes a falhas. Qualquer mudança em valores ou estrutura do grafo produz um novo grafo. Um grafo de propriedades corresponde a um par de RDDs que codifica as propriedades de cada vértice e aresta e possui membros para acessar seus vértices (*vertices*) e arestas (*edges*).

Grafos podem ser construídos de diversas e diferentes formas, como com base em arquivos de dados, RDDs e até mesmo de geradores sintéticos. No exemplo acima, podemos ver a criação de RDDs para armazenar os vértices utilizados para representar os usuários de um sistema. Essas estruturas de dados podem ser observadas na Figura 15:

Figura 15 – Criação de RDDs

// Assumindo que o SparkContext já está definido
val sc: SparkContext
// Cria um RDD para os vértices
val usuarios: RDD[(VertexId, (String, String))] =
  sc.parallelize(Array((3L, ("Rafael", "estudante")), 
                       (7L, ("João", "posdoutorado")),
                       (5L, ("Fernando", "prof")), 
                       (2L, ("Ingrid", "prof"))))
// Cria um RDD para as arestas
val relacionamentos: RDD[Edge[String]] =
  sc.parallelize(Array(Edge(3L, 7L, "Colaborador"),
                       Edge(5L, 3L, "Orientador"),
                       Edge(2L, 5L, "Colega"), 
                       Edge(5L, 7L, "Pesquisador")))
// Define um usuário padrão, caso exista uma relacionameto
sem usuário definido
val defaultUser = ("John Doe", "Missing")
// Constrói o grafo inicial
val graph = Graph(usuarios, relacionamentos, defaultUser)


Figura 16 – Exemplo de grafo

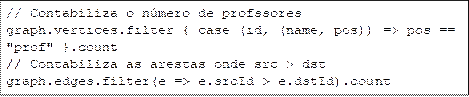


Fonte: Spark Apache, S.d.b.

Além disso, no exemplo da Figura 15, vemos o uso da classe Edge para representar as arestas. O construtor da classe Edge recebe os parâmetros srcId e dstId que correspondem aos identificadores dos vértices de origem e destino. Além disso, outro parâmetro é o atributo que armazena a propriedade da aresta. Continuando a observar o exemplo da Figura 15, podemos desmembrar o grafo em arestas e vértices por meio do uso respectivamente dos membros *graph.vertices* e *graph.edges*, da seguinte forma:

Vale destacar que o membro *graph.vertices* retorna um *VertexRDD[(String,String)]*, que é uma extensão do objeto *RDD[(VertexId,(String,String)]*. Por outro lado, o *graph.edge* retorna um *EdgeRDD*, que contém um objeto *Edge[String]*.

Figura 17 – Graph.vertices e graph.edges



Além disso, a API GraphX estende a classe *Edge* ao adicionar os membros *srcAttr* e *dstAttr*, contendo, respectivamente, as propriedades dos vértices de origem e destino. Assim é definida a classe *EdgeTripplet* que une de forma lógica as propriedades de arestas e vértices. Dessa forma, podemos realizar operações sobre os dados.

Figura 18 – Classe *EdgeTripplet*

class Graph[VD, ED] {
  val vertices: VertexRDD[VD]
  val edges: EdgeRDD[ED]
}


Assim como os RDDs possuem operações básicas como *map*, *filter* e *reduceByKey*, grafos de propriedades (ou Property *Graphs*) também possuem uma coleção de operadores básicos que aceitam funções definidas por usuário e produzem novos grafos com propriedades e estruturas transformadas. Os principais operadores que possuem implementações otimizadas são definidos pela classe *Graph* e operadores convenientes, que são expressos como uma combinação das principais operações são definidas pela classe *GraphOps*. Graças a características da linguagem Scala, os operadores em *GraphOps* estão disponíveis como membros de *Graph*.

**FINALIZANDO**

Nesta aula, aprendemos sobre o sistema de computação em *cluster* Spark, sistema que pode ser utilizado em conjunto com o ecossistema Hadoop ou de forma independente com o auxílio de outras ferramentas. Além disso, esta aula trouxe uma visão um pouco mais aprofundada a respeito dos módulos suportados pela plataforma.

No primeiro tema, nos aprofundamos no funcionamento básico da plataforma Spark, também conhecida por *Spark Core*. Além disso, aprendemos os principais conceitos da arquitetura Spark, SparkContext, sua principal estrutura de dados, os RDDs, e como utilizar suas operações.

Em seguida, aprendemos a maneira que o ecossistema Spark utiliza para trabalhar com dados estruturados. Conhecemos os principais conceitos envolvendo o Spark SQL. Outro assunto importante é a implementação de novas estruturas de dados otimizadas para tratar os dados estruturados: *datasets* e *dataframes*.

 O terceiro tema nos apresentou a plataforma de processamento e a ingestão de fluxos de dados do ecossistema Spark. Procuramos nos aprofundar no funcionamento da arquitetura do Spark *streaming* e entendemos os conceitos de fluxos discretizados e janelas de transformações. Além disso, desde o tema anterior, pudemos entender a interface de processamento de fluxos de dados estruturados que utilizam as estruturas de dados definidas no Spark SQL em conjunto com a arquitetura de processamento em *microbatch* do Spark *streaming*.

Na sequência, conhecemos a biblioteca de aprendizado de máquina (ou *machine learning*) MLLib. Primeiramente, entendemos os principais conceitos e algoritmos que são utilizados no aprendizado de máquina e aprendemos como funciona um fluxo de aprendizado de máquina. Em seguida, conhecemos os principais componentes que compõem a biblioteca MLLib e como funciona a criação de *pipelines* de aprendizagem de máquina.

Por fim, no último tema de nossa aula, conhecemos o componente de processamento de grafos em sistemas distribuídos de larga escala. Primeiro revisitamos a definição dos principais conceitos da teoria dos grafos e os principais algoritmos utilizados no processamento em sistemas distribuídos. Em seguida, conhecemos os principais componentes da API GraphX.

**REFERÊNCIAS**

CHAMBERS, B; ZAHARIA,M.**Spark – The definitive Guide:**Big Data processing made simple. Sebastopol CA: O’Reilly Media, inc., 2018

CLUSTER Mode Overview. **Spark Apache**, S.d. Disponível em: <https://spark.apache.org/docs/latest/cluster-overview.html>. Acesso em: 25 nov. 2020.

KARAU, H; KONWINSKI, A; WENDELL, P; ZAHARIA, M.**Learning Spark: lightning-fast data analysis**. Sebastopol CA: O’Reilly Media, inc., 2015.

ML pipelines. **Spark Apache**, S.d. Disponível em: <https://spark.apache.org/docs/latest/ml-pipeline.html>. Acesso em: 25 nov. 2020.

SPARK SQL, Dataframes and datasets Guide. **Spark Apache**, S.d. Disponível em: <https://spark.apache.org/docs/latest/sql-programming-guide.html>. Acesso em: 25 nov. 2020.

SPARK APACHE. **Spark streaming programming Guide**. Wilmington, DE: The Apache Software Foundation, S.d.a. Disponível em: <https://spark.apache.org/docs/latest/streaming-programming-guide.html>. Acesso em: 25 nov. 2020.

\_\_\_\_\_. **GraphX programming Guide**. Wilmington, DE: The Apache Software Foundation, S.d.b. Disponível em: <https://spark.apache.org/docs/latest/graphx-programming-guide.html>. Acesso em: 25 nov. 2020.

**BIG DATA**

AULA 6

Prof. Luis Henrique Alves Lourenço

**CONVERSA INICIAL**

Nesta aula, será apresentado a você um aprofundamento em alguns temas complementares aos estudos sobre *big data*. Inicialmente, serão apresentadas algumas tecnologias que podem não estar diretamente relacionadas ao Hadoop, mas que podem servir como alternativas muito interessantes e que podem ser encontradas no desenvolvimento de diversas aplicações de *big data* existentes. Em seguida, vamos explorar a abordagem de gerenciamento de dados armazenados em formato natural, conhecida como *data lake*. No tema seguinte, aprofundaremos o desenvolvimento de sistemas de recomendação utilizando *big data*. No quarto tema de nossa aula, conheceremos alguns detalhes sobre as plataformas de computação em nuvem. E, por fim, exploraremos os principais pontos no desenvolvimento de projetos de arquitetura de *big data*.

**TEMA 1 – OUTRAS TECNOLOGIAS**

Já conhecemos em detalhes muitas das principais tecnologias do ecossistema Hadoop, no entanto, podemos destacar mais algumas aplicações que podem ser muito úteis para o desenvolvimento de soluções de *big data*.

**1.1 IMPALA**

Impala é um motor de consultas SQL mantido pela Fundação Apache para executar em *clusters* Hadoop e que oferece a capacidade de atuar como banco de dados analítico massivamente paralelo. Entre as suas principais características destaca-se a possibilidade de realizar consultas SQL de baixa latência de dados armazenados em HDFS e HBase sem exigir movimentação ou transformação de dados. Essa ferramenta foi projetada para consultas analíticas em Haddop utilizando SQL ou ferramentas de *business inteligence* (BI). Com isso, o Impala é capaz de realizar processamento de dados em larga escala e consultas interativas no mesmo sistema através dos mesmos dados e metadados, ou seja, essas operações podem ser realizadas sem a necessidade de migrar conjuntos de dados para sistemas especializados ou formatos proprietários apenas para efetuar a análise dos dados.

A decisão de arquitetura do Impala de acessar os dados em HDFS ou HBase diretamente tem como consequência a redução de latência. Assim, é possível obter desempenho muito maior do que aplicações semelhantes como o Hive, dependendo do tipo de consulta e configuração. Uma vez que o processamento é realizado localmente em cada nó, gargalos de rede são evitados. Isso permite que toda operação ocorra sem a conversão de formatos de dados e, com isso, todos os dados podem ser consultáveis sem atrasos.

**1.2 ACCUMULO**

O Accumulo é um projeto da Fundação Apache inspirado no sistema de armazenamento distribuído BigTable, do Google, ou seja, seu modelo de dados é baseado em armazenamento de chave-valor. O Accumulo foi projetado para utilizar o HDFS para armazenar os dados. Uma das principais características do Accumulo é a segurança em nível de célula, pois cada par chave-valor possui o seu próprio rótulo de segurança que tem a capacidade de limitar os resultados de uma consulta baseado nas autorizações de acesso do usuário. Dessa forma, é possível armazenar dados com diferentes requisitos de segurança na mesma tabela. Outra característica importante é o mecanismo de programação capaz de modificar pares chave-valor durante processos de gerenciamento de dados através do uso de iteradores (*SortedKeyValueIterators*), que são abstrações que permitem a implementação de operações diretamente no conjunto de servidores que armazena os dados (*TabletServers*). Desse modo, é possível realizar operações de transformações nos dados assim que são inseridos.

**1.3 REDIS**

Redis é um sistema *open source* de armazenamento de dados em memória utilizado como banco de dados, cache e agente de mensagens (*message broker*). Seu nome é um acrônimo de *REmote DIctionary Server* (servidor de dicionário remoto). O Redis possui uma arquitetura mestre-subordinado que replica os dados de forma assíncrona por vários servidores. O sistema de armazenamento do Redis é baseado em uma estrutura de dados de chave-valor em que os dados se localizam na memória do servidor. Dessa forma, é possível reduzir o tempo de busca, uma vez que a quantidade de acessos a disco é reduzida. No entanto, o Redis também é capaz de fornecer o armazenamento de dados em disco como forma de garantir a persistência dos dados e sua rápida recuperação em caso de uma interrupção.

O Redis foi projetado inicialmente para ser um cache na memória para reduzir o tempo de acesso a bancos de dados, porém, suas características o fazem uma aplicação extremamente versátil.  A estrutura de dados chave-valor permite que chaves façam o mapeamento para dados em diversos tipos, podendo ser dados de texto ou binários limitados a 512 MB. Além disso, os dados no Redis podem ser configurados com tempo de vida (TTL), ou seja, cada chave pode ser configurada para ser removida automaticamente depois de determinado período de tempo. Isso permite que o Redis seja bastante utilizado como gerenciador de sessões. Além disso, ele também é utilizado para a implementação de chat e sistemas de mensagens, filas, classificações em tempo real ou limitadores de taxas.

**1.4 IGNITE**

Ignite é um projeto mantido pela Fundação Apache que pode ser definido como uma plataforma de computação em memória escalável, distribuída e tolerante a falhas para aplicações de tempo real que pode processar *terabytes* de dados em velocidade de memória. Uma de suas principais características é a capacidade de atuar tanto como uma camada de cache quanto como um sistema de armazenamento completo. Isso quer dizer que o Ignite é um sistema de computação em memória com a capacidade de armazenar dados em disco. Dessa forma, ele pode atuar como um banco de dados em memória, um *data grid* em memória ou como um banco de dados escalável horizontalmente. O Ignite pode ser entendido como um substituto do Redis, porém a principal diferença é que o Ignite possui características mais voltadas a um banco de dados, como a garantia das propriedades ACID, suporte à SQL.

**1.5 NIFI**

NiFi é um sistema de processamento e distribuição de dados que utiliza grafos dirigidos para automatizar e gerenciar fluxos de dados entre sistemas mantido pela fundação Apache. Dessa forma, o NiFi foi projetado para enfrentar os desafios que envolvem plataformas de processamento baseado em fluxos. Entre os seus pontos mais fortes, destacam-se o oferecimento de uma interface web para projetar, controlar e monitorar fluxos de dados. O NiFi permite configurar o nível de perda de dados, garantia de entrega, latência, vazão, priorização dinâmica, alteração de fluxos, segurança, entre outras características. Alguns dos principais conceitos do Nifi incluem:

* ***FlowFile*:** é o pacote de informação (*Information Packet*) que representa os objetos de dados que se movimentam entre os sistemas.
* ***FlowFile processor*:** são os elementos processadores de fluxos de dados que realizam operações como roteamento de dados, transformação ou mediação entre sistemas.
* **Conexão:** são os elementos que fazem a ligação entre os processadores de fluxos de dados. Atua como uma espécie de *buffer* ou fila, que pode ser utilizada para priorizar certos dados dinamicamente.
* ***Flow controller*:** é um *scheduler* utilizado para definir a ligação entre os elementos do fluxo. Atua como um intermediário para facilitar a troca de *FlowFiles* entre os processadores de fluxos de dados.

***Process group*:** é um subconjunto de processadores de fluxos de dados e conexões capazes de receber dados e produzir resultados.

**1.6 AMBARI**

Ambari é uma ferramenta que fornece uma interface Web fácil e intuitiva baseada em uma API REST para o provisionamento, gerenciamento e monitoramento de *clusters* Hadoop capaz de realizar a integração do Hadoop em uma infraestrutura já existente. Em sua interface, o Ambari exibe diversas ferramentas para monitorar o seu *cluster* Hadoop. Por exemplo, é possível monitorar a capacidade de armazenamento do *cluster*, a quantidade de nós ativos, a utilização de memória, rede e CPU, a carga do *cluster*, entre muitos outros detalhes. Além disso, o Ambari fornece uma interface para realizar a ativação, a desativação, o gerenciamento e utilização de diversos dos componentes de um cluster Hadoop. Para isso, o Ambari divide os componentes do Hadoop em três categorias:

* ***Core Hadoop*:** componentes básicos do Hadoop, como o HDFS e o MapReduce.
* ***Essential Hadoop*:**componentes projetados para facilitar o trabalho com os componentes do *Core Hadoop*. Inclui componentes como: Pig, Hive, HBase e Zookeeper.
* ***Hadoop Support*:**conjunto de componentes que permitem o gerenciamento e o monitoramento de uma instalação Hadoop e permite conectar o Hadoop a um *cluster*.

**TEMA 2 – *DATA LAKE***

Já sabemos que a maior parte do tempo gasto em um projeto de análise de dados se concentra nas tarefas que envolvem o gerenciamento de dados tais como a identificação, limpeza e integração dos dados. Segundo Chessell et al. (2014), isso ocorre por três motivos principais:

* Os dados estão espalhados em diversas aplicações e sistemas de negócios;
* Os dados frequentemente precisam ser reorganizados e reformatados para serem mais facilmente analisados;
* Os dados devem ser atualizados frequentemente para manter a sua relevância.

Diferentemente de outras estratégias de armazenamento de dados como *data warehouse* e *data mart*, que consistem em dados extraídos de sistemas transacionais, ou seja, são formados por dados estruturados, como já sabemos, os dados que utilizamos no processamento de *big data* podem ser estruturados em diferentes formatos, semiestruturados ou nem precisam ser estruturados, pois são dados que podem vir de sistemas completamente diferentes decorrentes de necessidades muito diferentes. Portanto, esses modelos de armazenamento de dados não se mostram apropriados para tais aplicações. Por isso, é muito difícil a manutenção de sistemas de obtenção de dados para análise e, portanto, se mostra muito necessária para a criação de uma estrutura de governança de dados que possa incorporar dados processados ou não de forma que possam ser utilizados por ferramentas de *big data* para a geração de informações valiosas através da análise de tais dados.

O termo *data lake* foi inventado e descrito pela primeira vez por James Dixon em seu blog, em 2010, da seguinte forma:

Se você pensar em um *data mart* como uma loja de garrafas de água (água limpa, embalada e estruturada para ser consumida facilmente), o *data lake* seria um corpo de água maior em estado natural. O conteúdo do *data l*flui a partir de uma fonte que preenche o lago e vários usuários do lago podem vir e examinar, mergulhar ou colher amostras. (Dixon, 2010)

Uma vez que se trata de dados em seu estado natural para uma comunidade de usuários, o conceito de *data lake* se mostra muito mais adequado a soluções de *big data*, nas quais um *data lake* corresponde a um conjunto de dados de diversas fontes que são armazenados sem que sejam transformados e onde os dados só serão transformados e terão esquemas aplicados para atender às necessidades de sua análise. Um *data lake* pode, então, receber quaisquer dados e seus dados podem ser livremente atualizados.

No entanto, sem um gerenciamento adequado, os dados podem se degradar a um estado onde não existe mais como saber a sua confiabilidade, fazendo com que percam seu valor e o *data lake* se torne em um tipo de pântano (*data swamp*) que não permite mais a extração de informações úteis. Para que o valor dos dados contidos em um *data lake* não seja perdido, faz-se necessário criar uma solução de *data lake* que inclua as noções de gerenciamento, acessibilidade e governança. Para muitos, essa solução é uma adaptação de *data lake* chamada de *data reservoir* (ou reservatório de dados) (Chessell et al., 2014). Um *data reservoir* pode ser compreendido como um *data lake* capaz de catalogar e proteger os seus dados.

 Dessa forma, a estratégia de armazenamento de dados *data lake* pode oferecer uma redução de complexidade, que as estratégias *data warehouse* e *data mart* não são capazes de oferecer, uma vez que, para atender à mesma demanda que um único *data lake* é capaz de atender, faz-se necessária a utilização de diversos desses sistemas. Isso também gera impacto no custo e eficiência desses sistemas. Além disso, *data lakes* com bons esquemas de governança são capazes de trazer mais transparência para o uso dos dados.

Um dos principais conceitos que envolvem a governança de dados é o conceito de metadado que, em outras palavras, são as informações que se podem obter a respeito dos dados em si. Ou seja, são os dados que os sistemas de *data lake* utilizam para gerenciar os seus dados e permitem a governança de dados. Os metadados podem ser divididos em três categorias:

* **Metadados técnicos:** são os metadados que fornecem informações a respeito da forma ou estrutura dos dados, como tamanho, tipo de dado, esquema.
* **Metadados operacionais:** são os metadados que fornecem informações sobre a cadeia de fornecimento dos dados como qualidade, perfil, proveniência dos dados, e a sua linhagem, ou seja, sua cadeia de fornecimento.
* **Metadados de governança (ou de negócio):** são os metadados que fornecem a terminologia de negócio de cada dado, ou seja, são as informações a respeito do que os dados representam para o usuário final e que podem ser utilizadas para tornar mais fácil encontrar e entender tais dados, como *tags*, descritores, nomes de negócios, qualidade e regras de mascaramento.

Uma vez que os metadados são armazenados, eles permitem a possibilidade de buscar, localizar e aprender sobre os dados disponíveis no *data lake*. Para isso, é necessário gerenciar uma camada adicional para armazenar e catalogar os metadados. Essa é a primeira garantia que impede que o seu *data lake* se torne um *data swamp*. Dessa forma, é necessário que todos os dados que sejam inseridos no *data* *lake* passem por um processo de criação e extração de metadados.

**2.1 MATURIDADE DE *DATA LAKE***

A importância de se trazer dados em seu formato natural para o sistema de armazenamento é que esses dados podem ser utilizados para análise de autoatendimento (*self-service analytics*), que é uma abordagem de análise mais avançada, em que os consumidores dos dados possuem a habilidade de explora-los diretamente. Essa análise possui grandes desafios no que se refere à governança e à segurança de dados. A governança da informação de um projeto de *big data* (ou seja, a atitude tomada em relação ao gerenciamento de seus dados) se manifesta diretamente nos resultados obtidos por ele, uma vez que permitem garantir segurança, organização, catálogo das informações e acessibilidade aos dados. No entanto, o grande volume que o *big data* pode atingir nos mostra que não é prático aplicar um conjunto rígido de processos sobre todo o conjunto de dados. Para isso, Gorelik (2019) define alguns estágios de maturidade de um *data lake*:

* ***Data puddle* (ou poça de dados)**: estágio inicial onde os dados de um *data mart* são utilizados em conjunto com tecnologia *big data*. Normalmente, é o primeiro passo na adoção de uma estratégia de Big Data dentro de uma organização. Essas estruturas possuem um escopo limitado e possuem o propósito de atender às necessidades de um time ou um projeto específico, muitas vezes servindo para atender a uma demanda por processos que exigem um uso intenso de computação e dados. Ou ainda servindo de local de testes para cientistas de dados. Sua construção e manutenção exigem um alto envolvimento de uma equipe técnica.
* ***Data ponds* (ou lagoa de dados)**: estágio em que a organização possui uma coleção de data *puddles*, ou seja, na prática funciona como um *data warehouse* mal projetado, com diversos *data marts* para propósitos específicos, que utilizam tecnologia *big data*, porém, comparado com *data* *puddles*, utilizam tecnologias mais baratas e escaláveis que *data* *warehouses* e *data marts* relacionais. Não possuem muitas vantagens, uma vez que continuam com limitações semelhantes aos *data puddles* e não colaboram para melhorar o acesso aos dados, nem para a tomada de decisões de negócio baseadas em dados.
* ***Data lake*** **(ou lago de dados)**:difere do *data pond* no sentido em que suportam análise de autoatendimento, ou seja, os usuários são capazes de realizar buscas e localizar dados diretamente. Além disso, são capazes de armazenar dados que não possuem propósito em nenhum projeto atual, mas que podem ser úteis para usuários de negócios.
* ***Data ocean*:**o oceano de dados expande o conceito de *data lake* para toda uma organização. Dessa forma, todos os dados da organização estão disponíveis para análise de autoatendimento e decisões estratégicas de negócios baseados em dados.

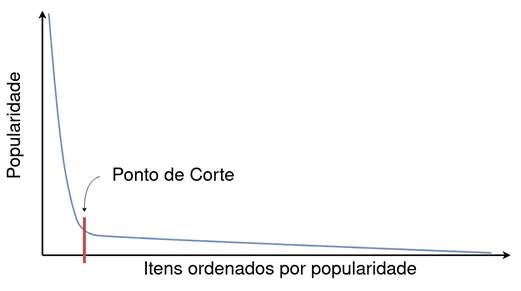
**TEMA 3 – SISTEMAS DE RECOMENDAÇÃO**

Sistemas de recomendação são uma classe de aplicações que buscam prever a reação dos usuários em relação a um conjunto de opções. Dessa forma, esses sistemas são capazes de fornecer uma experiência personalizada para cada usuário, muitas vezes com o intuito de oferecer opções mais adequadas. Para isso, esses sistemas buscam encontrar relações entre suas distintas partes, seja por meio da comparação de ações passadas de forma a permitir uma predição de quais são as ações futuras mais prováveis, seja por meio da comparação entre as características de cada elemento.

Atualmente, qualquer serviço que ofereça acesso a conteúdo pode possuir um catálogo imenso. Em muitos casos, os usuários já sabem de antemão o que desejam. Dessa forma, o usuário pode acessar o conteúdo por meio de uma simples busca pelo catálogo, porém, em outros casos, os usuários não sabem exatamente o que querem e é aqui que os sistemas de recomendação se tornam necessários. Assim, o sistema recomenda ao usuário certos produtos que ele acredita ser de interesse baseado nas informações que ele sabe do usuário. Então, podemos dizer que a necessidade de sistemas de recomendação surgiu devido à dificuldade em divulgar catálogos imensos de forma mais eficiente. O desafio aqui pode ser explicado através do fenômeno da cauda longa (*the long tail*). Esse fenômeno é baseado em uma característica conhecida de algumas distribuições estatísticas que diz que a maioria das ocorrências correspondem a uma minoria dos itens na distribuição.

Lojas físicas possuem uma limitação de quantos produtos podem possuir em suas prateleiras, por isso precisam escolher quais produtos deseja expor para seus usuários. Nesse cenário, a recomendação é bastante simples: uma vez que o vendedor não é capaz de conhecer suficientemente a todos os usuários, a escolha de quais produtos devem ser expostos se baseia em números agregados. Ou seja, para maximizar suas vendas, o lojista deve escolher expor os produtos mais populares em suas prateleiras. Dessa forma, o fenômeno da cauda longa é capaz de distinguir sistemas limitados como os das lojas físicas dos sistemas on-line que não possuem uma limitação de espaço para exibir seus produtos. Na Figura 1, o gráfico representa o fenômeno da cauda longa em que o eixo vertical representa a popularidade (a quantidade de vendas) dos produtos, representados de forma ordenada no eixo horizontal. Nele podemos observar que o volume de vendas está concentrado em uma parte reduzida dos produtos e que há um ponto de corte onde a partir do qual não há viabilidade para o lojista explorar. Essa parte da curva é denominada de *cauda longa*.

Figura 1 – Cauda longa



Fonte: Leskovec; Rajaraman; Ullman, 2010.

Por outro lado, os sistemas on-line não possuem tal limitação e podem explorar toda a faixa de itens. Por essa razão, o catálogo de sistemas on-line costuma ser muitas vezes maior que o catálogo de lojas físicas. No entanto, a existência de tantos itens é um desafio para qualquer usuário encontrar os itens de seu interesse. Dessa forma, podemos dizer que mais opções de escolha exigem melhores filtros. Além disso, vale a pena observar que esses sistemas não são úteis apenas para sistemas de venda ou recomendar músicas, filmes e podem ser utilizados em redes sociais para selecionar a lista de postagens que aparecerá no *feed* de cada usuário e, até mesmo, os motores de busca podem funcionar como sistemas de recomendação, uma vez que passem a retornar buscas direcionadas a determinado perfil de usuário. Esses sistemas têm em comum a característica de proporcionar ao usuário uma experiência personalizada, ou seja, eles levam em consideração a relação entre as características ou comportamentos do usuário em relação aos itens com que interage.

**3.1 TIPOS DE RECOMENDADORES**

Os sistemas de recomendação podem ser divididos em grupos baseados em como são construídos:

* **Editoriais ou curadorias:** são as listas de recomendações criadas à mão. Normalmente, levam em consideração os interesses de quem está criando, por exemplo, uma lista de favoritos. Não levam em consideração nenhuma característica do usuário.
* **Agregações simples:** listas que agregam alguma característica do próprio conteúdo. Por exemplo: *Top 10*, listas de popularidade, mais recentes. Essas listas podem até levar em consideração algum fator relacionado com seus usuários, por exemplo, a soma de interações que a totalidade de usuários tem com o conteúdo (vendas, *views* etc.).
* **Recomendadores individualizados:** são as recomendações que levam em conta as características dos usuários e dos produtos para gerar uma experiência personalizada para cada usuário. Aqui podemos elencar os sistemas de recomendação que nos interessam nesse tema.

Os recomendadores individualizados utilizam uma função de utilidade que recebe um usuário e um item e os mapeia a uma avaliação. Dessa forma, é possível criar uma matriz de utilidade com a avaliação que cada usuário faz de cada item. Obviamente trata-se de uma matriz esparsa, uma vez que, na maioria dos casos, os usuários avaliam apenas uma parcela muito pequena do conjunto de itens disponíveis. Existem duas formas de se obter avaliações de usuários. A primeira delas são as avaliações explícitas, que é quando o usuário ativamente determina uma nota de avaliação para o item. No entanto, apenas uma minoria dos usuários realiza avaliações, logo não escalam muito bem. A outra forma são as avaliações implícitas em que o sistema determina as avaliações de acordo com o comportamento do usuário.

Uma vez que temos avaliações que representam a relação entre cada usuário e cada item, podemos obter a matriz de utilidade. Ela é muito importante para os sistemas de recomendação, pois as recomendações se baseiam nas previsões de preenchimento das avaliações desconhecidas. Existem duas principais abordagens para isso: recomendações baseadas em conteúdo (*content-based recommendations*) e filtragem colaborativa (*collaborative filtering*).

**3.2 CONTENT-BASED RECOMMENDATIONS**

A ideia principal dessa abordagem é recomendar ao usuário itens parecidos, ou seja, com características semelhantes aos que ele avaliou. Dessa forma, o foco dessa abordagem está nas características dos itens. A similaridade entre diferentes itens está baseada na similaridade de suas características. Portanto, é importante construir um perfil para cada item que represente suas características e possa ser comparado para fins de calcular a similaridade com outros perfis de itens. É conveniente pensar em um perfil como um vetor de características. Da mesma forma, o perfil de um usuário é construído baseado em suas interações com os itens. Podemos pensar que os perfis de usuários são calculados por uma soma ponderada (levando em consideração a avaliação do usuário) dos perfis de itens avaliados. Assim, é possível realizar predições de avaliação, ou seja, é possível inferir qual será a avaliação de um item para determinado usuário através de seus perfis.

As vantagens de usar essa abordagem incluem a não dependência entre usuários, o que significa que não são necessários dados de outros usuários para recomendar itens a um usuário. Isso também indica que é possível fazer recomendações para usuários com gostos específicos. Além disso, não são necessárias avaliações prévias. Um usuário pode receber recomendações de um item novo assim que ele é inserido e também permite que a recomendação possa ser explicada, uma vez que se conhecem as características que levaram até ela.

As desvantagens incluem a dificuldade em encontrar as características adequadas. Não é uma tarefa simples definir quais são as melhores características para descrever um tipo de item. Outra desvantagem é a superespecialização, ou seja, itens fora do perfil de conteúdo do usuário nunca são recomendados. Múltiplos interesses são ignorados, assim como não levam em consideração a popularidade desses itens. Por fim, existe a dificuldade de criar perfis para novos usuários, uma vez que estes não avaliaram nenhum item.

**3.3 COLLABORATIVE FILTERING**

A abordagem de filtragem colaborativa (ou *collaborative filtering*) parte do pressuposto de que usuários com avaliações semelhantes podem ser utilizados para estimar a avaliação que um deles fará de determinado item. Para isso é necessário calcular a similaridade entre os usuários, ou seja, o quanto as suas avaliações são parecidas. Algumas estratégias podem ser utilizadas para isso, porém a que apresenta os melhores resultados é a *Pearson Correlation*. Essa forma de calcular a similaridade entre dois perfis de usuários utiliza o cálculo do cosseno entre dois perfis normalizados pela subtração da média para cada usuário. Dessa forma, a avaliação média de cada usuário se torna zero, eliminando a distorções. Levando-se em conta o cálculo de similaridades de perfis de usuários, podemos realizar a predição de avaliações para dado usuário através da média das avaliações do conjunto de usuários mais semelhantes a ele ponderada pela soma das similaridades do conjunto de usuários semelhantes.

Podemos aplicar o mesmo princípio para as avaliações entre itens (ou *Item-Item collaborative filtering*). Dessa forma, para estimar a avaliação que determinado usuário poderá fazer para determinado item, deve-se buscar por itens avaliados pelo usuário com avaliações semelhantes. Para isso, as mesmas funções de métrica e predição do modelo de avaliação entre usuários (*User-User collaborative filtering*). Ambos os modelos de avaliação possuem abordagens complementares. Na prática, o modelo de avaliação entre itens possui melhores resultados para a maioria dos casos, pois itens são relativamente mais simples que usuários, portanto, a similaridade de itens é mais significativa que a de usuários.

Além disso, a grande vantagem da abordagem de filtragem colaborativa é que pode ser aplicada a qualquer tipo de item e não é necessária nenhuma seleção de características. Enquanto que o ponto fraco de tal abordagem é a necessidade de uma quantidade mínima de usuários para comparar similaridades. Os dados são esparsos, ou seja, é difícil encontrar usuários que avaliaram os mesmos itens e não é possível recomendar um item que nunca foi avaliado. Por fim, temos de considerar o viés de popularidade, ou seja, a tendência de que os itens mais populares sejam mais recomendados por serem normalmente melhor avaliados.

**3.4 MÉTODOS HÍBRIDOS**

É possível adicionar métodos de uma abordagem em outra abordagem, por exemplo, ao adicionar métodos baseados em conteúdo (*content-based*) em uma filtragem colaborativa. Dessa forma, é possível criar perfis de item para resolver o problema de novos itens que nunca foram avaliados em uma filtragem colaborativa. Além disso, é possível implementar mais de um recomendador diferente e combinar as suas predições utilizando, por exemplo, abordagens que empregam modelos lineares, por exemplo, o uso de informações demográficas para lidar com o problema de perfis de novos usuários.

**3.5 AVALIAÇÃO DE SISTEMAS DE RECOMENDAÇÃO**

Sistemas de recomendação podem ser avaliados por meio da remoção de uma parte do conjunto de dados da matriz de utilidade. Dessa forma, os valores removidos são calculados pelo sistema de recomendação. Em seguida, os dados estimados podem ser comparados com os dados conhecidos que foram removidos da matriz de utilidade por meio do cálculo da raiz do erro médio quadrático (RMSE, em inglês). A RMSE é uma medida muito utilizada para aferir a qualidade de um modelo e funciona como uma medida análoga ao desvio padrão, sendo capaz de explicitar a diferença entre a predição e o valor real.

**TEMA 4 – COMPUTAÇÃO EM NUVEM**

Como vimos em nossos estudos, a metodologia de processamento de *big data* permitiu a captura, o armazenamento e a análise de imensas quantidades de dados. Além disso, a arquitetura Hadoop permitiu a implementação de clusters em máquinas comuns. Com isso, podemos analisar dados de muitas fontes que antes eram ignoradas como *logs*, dados de sensores, informações de redes sociais e muitos outros dados que não se encontram armazenados em bancos de dados relacionais. A análise de dados não está mais limitada aos dados estruturados, e agora pode incluir um volume de dados cada vez maior. Aliado a isso, a tecnologia dos equipamentos de armazenamento de dados continua evoluindo e, com isso, os preços do *gigabyte* armazenado seguem a tendência de queda. No entanto, o volume de dados sendo produzidos e que podem ser incluídos para análise tem crescido tão rápido que mesmo todo esse desenvolvimento de tecnologias mais baratas para processar e armazenar cada vez mais dados não tem sido suficiente para que muitas empresas pequenas e médias possam adotar essas tecnologias.

Dessa forma, surge como uma opção viável para essas empresas a adoção de sistemas de *cloud computing* (ou computação em nuvem) para armazenar e processar seus dados utilizando tecnologia de *big data*. Nesse contexto, *cloud computing* representa o acesso de rede sob demanda a recursos computacionais fornecidos por uma entidade externa. Ou seja, a *cloud computing* permite o acesso a serviços de computação escaláveis e elásticos por meio da internet. Esses serviços podem oferecer às empresas uma redução de infraestrutura, bem como uma redução nos custos com licenças e manutenção de softwares específicos, além de permitir um ambiente viável para a implementação de tecnologias que fazem uso do processamento de *big data* em empresas de pequeno e médio porte. Também devemos destacar que esses sistemas podem servir como forma de testar novas tecnologias antes de a empresa optar por fazer um investimento maior. Por outro lado, a preocupação com a segurança e o controle dos dados são pontos de discussão.

Quando uma empresa investe em uma infraestrutura de *cloud computing* interna à própria organização, essa infraestrutura é classificada como privada, em contraposição às infraestruturas de *cloud computing* públicas, que são aquelas abertas ao uso por organizações externas.

**4.1 MODELOS DE SERVIÇO DE *CLOUD COMPUTING***

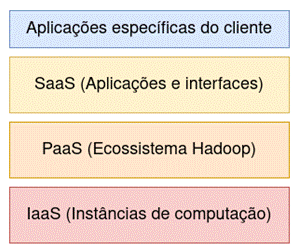
Uma vez que a tecnologia de *cloud computing* se propõe a oferecer serviços para atender às demandas de computação de entidades externas, podemos classificar os serviços oferecidos em categorias como:

* **PaaS (*platform as a service*):** a plataforma como um serviço pode ser entendida como o uso de *cloud computing* para oferecer plataformas para o desenvolvimento e uso de aplicações customizadas necessárias para analisar grandes quantidades de dados. Ou seja, são aplicações que oferecem soluções para o design de aplicações, testes de aplicação, versionamento, integração, hospedagem, gerenciamento de estado, e demais ferramentas de desenvolvimento. Esse tipo de serviço pode reduzir os requisitos para o desenvolvimento de novos sistemas, além de permitir uma redução de custos e riscos.
* **SaaS (*software as a service*):** software como um serviço fornece às empresas aplicações que são armazenadas e executam em servidores virtuais. O custo de um serviço normalmente é cobrado por largura de banda, tempo de uso e número de usuários, e as empresas se beneficiam de menores esforços de administração e manutenção das aplicações, além da facilidade de acesso ao software em qualquer lugar. A principal diferença para o PaaS é que o modelo SaaS não oferece uma solução customizada enquanto que no modelo PaaS a empresa pode desenvolver soluções específicas às suas necessidades.
* **IaaS (*Infrastructure as a Service*):** a infraestrutura como um serviço fornece acesso ao uso de equipamentos para recursos computacionais como armazenamento, servidores, dispositivos de rede e outros tipos de hardware. Portanto, podemos afirmar que, de certa forma, todos os fornecedores de serviços de *cloud computing* o fazem sobre uma nuvem de IaaS. Além disso, esse modelo é muito utilizado para oferecer os serviços de recuperação de desastres, serviços de computação, serviços de armazenamento, serviços de *data center*, infraestrutura de *desktop* virtual, entre outros.

**4.2 BIG DATA AS A SERVICE**

Os modelos de *cloud computing* descritos anteriormente podem ser visualizados como camadas em um modelo de negócios. Em tal modelo, a camada mais básica seriam todos os serviços providos por um modelo IaaS, que incluem as instâncias de computação, hardware ou máquinas virtuais, armazenamento, rede e toda infraestrutura necessária. Em uma camada intermediária seriam encontrados os serviços de um modelo PaaS, ou seja, em tal camada se encontram as aplicações que fornecem uma plataforma para a execução de um software customizado para as necessidades do cliente tais como serviços de banco de dados, gerenciamento de cluster, distribuição e gerenciamento de processamento, entre outras aplicações. E em uma camada superior se encontram os serviços de um modelo SaaS, que são as aplicações genéricas que são utilizadas pelo usuário através de APIs, interfaces web.

Figura 2 – *Big data as a service*



Como podemos observar na Figura 2, um serviço de *big data* implementado sobre uma pilha de camadas de *cloud computing* naturalmente se encontra na camada de PaaS, uma vez que consideramos o Hadoop, ou qualquer outra tecnologia de processamento e armazenamento distribuído, como uma plataforma onde soluções são implementadas. Dessa forma, temos variações de BDaaS (*Big Data as a Service*) que podem implementar a camada de IaaS, a camada de SaaS ou ambas na mesma nuvem.

**4.3 PRINCIPAIS FORNECEDORES DE BDAAS**

Podemos destacar algumas das principais empresas no mundo que fornecem serviços de *big data* por meio de serviços públicos de *cloud computing*:

<li**Amazon EMR: o Amazon Elastic MapReduce é a ferramenta de *big data* da plataforma Amazon Web Services (AWS) que implementa as camadas IaaS e PaaS. Atualmente é a maior plataforma de *big data* em nuvem. O EMR é baseado no Hadoop e permite utilizar ferramentas como Spark, Hive, HBase, Flink, Presto, entre muitos outros serviços. O EMR processa big data utilizando clusters Hadoop hospedados nos servidores virtuais da nuvem pública Amazon Elastic Compute Cloud (EC2). Além disso, ele implementa uma infraestrutura de armazenamento distribuída própria, a Amazon Simple Storage Service (S3), que substitui o HDFS.**

* **Google Cloud Dataproc: é a ferramenta de *big data* do Google, que fornece serviços Spark e Hadoop, além de componentes como Hive, HBase, Zeppelin, Zookeeper, Presto, Pig, entre outros. Implementa uma combinação entre as camadas IaaS e PaaS. O Cloud Dataproc está integrado com outros serviços do Google Cloud Platform, como BigQuery, Cloud Storage, Cloud Bigtable, Stackdriver Logging e Stackdriver Monitoring.**
* **Microsoft Azure: implementa uma integração entre IaaS e PaaS, que fornece serviços hospedados nos servidores virtuais de *cloud computing* *Azure* *virtual machines*. O Azure também implementa serviços de armazenamento (*Azure* *blob storage*), CDN, serviço de *containers* baseados em *docker* (*Azure* *container services*), processamento em lote (*Azure* *batch*), computação sem servidor (*Azure* *functions*), e um serviço para permitir o uso e gerenciamento de *clusters* Hadoop e Spark (*Azure* *HDInsight*).**

**</li**

**TEMA 5 – DESIGN DE ARQUITETURA *BIG DATA***

Uma vez que conhecemos os conceitos que envolvem a arquitetura *big data*assim como as aplicações que implementam suas características, podemos pensar em como combinar todas essas tecnologias e ideias para o desenvolvimento de um produto visando atender a alguma necessidade existente. Devemos considerar também o impacto que tais tecnologias estão gerando na sociedade e nas empresas. Atualmente aplicações baseadas em *big data* representam um papel chave no ponto de vista dos negócios. No entanto, o desenvolvimento de aplicações de *big data* enfrenta dificuldades maiores que o desenvolvimento de sistemas baseados em tecnologias tradicionais. Esses desafios podem surgir da dificuldade de selecionar quais tecnologias de *big data* devem ser empregadas para cada tipo de necessidade. Além disso, também existem desafios no que se trata da complexidade em integrar sistemas de *big data*com os sistemas tradicionais existentes. Isso se expressa em algumas estimativas que dizem que mais da metade dos projetos de *big data* não conseguem ser postos em prática. As soluções de *big data* muitas vezes buscam objetivos como otimizar processos de negócios, adquirir vantagem competitiva, otimizar operações, entre muitos outros. Portanto, ao não conseguir concluir esses projetos, as empresas falham em obter uma grande variedade de resultados positivos.

No entanto, devido à complexidade de tal tarefa, não existe uma única maneira de implementar uma aplicação de *big data*. Dessa forma, precisamos das informações que serão utilizadas para guiar o desenvolvimento. Uma das maneiras de se obter tais informações é o entendimento detalhado das necessidades que a aplicação busca atender, ou seja, precisamos compreender questões tais como **quem serão os usuários**, **quais são os problemas**que a aplicação deve resolver, **quais os benefícios**mais importantes para os usuários, **como garantir**que sabemos o que os usuários querem e precisam e **como é a experiência**para os usuários. Dessa forma, é possível obter os requisitos-chave para o desenvolvimento.

Uma das abordagens que pode auxiliar na tarefa de criar uma boa documentação que seja capaz de levantar os requisitos corretos para o desenvolvimento de uma aplicação de *big data* evitando desperdícios e garantindo que a tudo que está sendo implementado esteja alinhado com as necessidades do usuário é a abordagem de desenvolvimento de produtos da Amazon conhecida como *Working Backwards*, ou trabalhando de trás para frente. Esse método é utilizado na empresa para o desenvolvimento de novos produtos focando no ponto de vista do usuário. Dessa forma, é possível obter o melhor entendimento sobre as necessidades do usuário e gerar documentação durante o levantamento de requisitos e antes de iniciar o desenvolvimento. Assim, todas as decisões de projeto acabam sendo tomadas de forma a otimizar a solução do problema real.

A ideia dessa abordagem consiste em começar pelo anúncio do produto, ou seja, pela divulgação de um *Press Release*. Apesar do nome, esse anúncio não precisa ser um comunicado de imprensa e vai depender do contexto de sua aplicação. Em muitos casos pode ser apenas uma comunicação interna da empresa para o desenvolvimento de uma ferramenta que será utilizada apenas internamente. O importante é que o produto seja anunciado na primeira fase para que os documentos de requisitos reflitam todas as críticas que o anúncio receber. Só depois que os requisitos estiverem sido obtidos é que podemos começar a fase de projetar o produto que será, então, desenvolvido. Dessa forma, o produto final tende a ser melhor ajustado às necessidades dos usuários.

**5.1 REQUISITOS DE APLICAÇÕES *BIG DATA***

Quando estamos analisando os requisitos dos usuários em relação a uma aplicação de *big data* temos que considerar diversos fatores. Primeiro, é importante destacar que devemos levar em consideração a infraestrutura e a experiência da equipe. É uma boa estratégia utilizar o conhecimento que a equipe já possui em vez de buscar uma tecnologia que seja nova para toda a equipe. Muitas vezes vale mais a pena fazer uma substituição de tecnologia para poder aproveitar algo que a equipe já possui afinidade do que a solução ideal, porém desconhecida pela equipe.

Como sabemos, um dos principais aspectos a se considerar em *big data* é o tamanho dos dados que o produto deverá manipular. Nesse ponto, o *Working Backwards* nos lembra que todas as decisões devem ser tomadas tendo como foco o ponto de vista do usuário. Dessa forma, não devemos utilizar uma tecnologia de *big data*, como o armazenamento distribuído se o volume de dados previsto não é tão grande, ou seja, não devemos tomar decisões de desenvolvimento com base na tecnologia que queremos adotar, mas devemos optar pela tecnologia que melhor atende ao cenário previsto. Deve-se tomar especial atenção ao tentar prever os requisitos para cenários futuros. No caso do armazenamento de dados, pode ser muito caro mover os dados para outro lugar, uma vez que o local de armazenamento foi definido. Além disso, é preciso definir quais serão as formas de ingestão de dados, ou seja, quais as tecnologias disponíveis para que os dados possam ser incorporados ao sistema.

Vale a pena destacar que a solução mais simples normalmente é a que vai atender melhor aos requisitos da aplicação. Escalar a complexidade de um sistema para atender a uma demanda além do previsto não é uma boa estratégia. Você pode aumentar os custos de manutenção sem que haja demanda para tal. Além disso, várias das tecnologias em *big data* são intercambiáveis e, dessa forma, podem ser substituídas facilmente. Tente sempre avaliar qual é o mínimo de infraestrutura necessária para operar a aplicação.

É importante lembrar que, em alguns casos, os dados devem ser mantidos armazenados no sistema por diversos motivos, como uma necessidade de realizar auditorias, ou por necessidades jurídicas, como é o caso de muitas aplicações financeiras. Dessa forma, pode ser necessária a implementação de algum tipo de política de retenção de dados, como é o caso de dados de navegação ou dados sensíveis de usuários. É interessante que sejam eliminados eventualmente por motivos de privacidade. Em casos como esses, é interessante haver uma avaliação prévia antes da implementação de uma estratégia de armazenamento de dados, para que os requisitos avaliados sejam atendidos pela política de governança dos dados. Além disso, também é importante definir uma política de expurgo, ou seja, procedimentos para a remoção de dados que não sejam mais necessários para nenhum propósito.

Também é importante ressaltar que as medidas de segurança adequadas devem ser tomadas para proteger os dados dos usuários. Vale sempre a pena verificar com o departamento jurídico da empresa se as regulamentações de segurança estão sendo cumpridas em relação à legislação de todos os países onde a aplicação estará em operação.

Outro aspecto que podemos levar em conta é a relação do sistema com o teorema CAP. Devemos avaliar quais as características do teorema que devem ser priorizadas, uma vez que o próprio teorema define que é impossível garantir os três atributos (consistência, disponibilidade e tolerância a partições) simultaneamente. A falta de disponibilidade em alguns sistemas pode custar muito dinheiro ou ser críticas a determinada aplicação. Para esses casos, deve se optar por um sistema de armazenamento que garanta que cada requisição por dados deve receber uma resposta, mesmo que não seja possível garantir que a resposta contém o dado atualizado. Da mesma forma, um sistema que depende que os dados requisitados sejam sempre os mais recentes, ou seja, que as transações não se percam e sejam executadas em uma ordem específica, devem priorizar soluções de armazenamento que garantam a consistência.

Outra questão muito importante trata dos padrões de acesso dos usuários em relação ao sistema. Neste ponto, precisamos entender qual é o tempo que o usuário precisa para ser atendido pela aplicação. É necessário avaliar quais soluções podem atender o usuário com um tempo de resposta compatível com sua expectativa. Por exemplo, para sistemas que precisam responder a requisições de usuários em milissegundos, pode ser necessária a adoção de bancos NoSQL como HBase ou Cassandra.

Os dados armazenados devem ser atualizados frequentemente para que as consultas possam ser úteis aos usuários. Dessa forma, é importante avaliar com qual frequência de tempo o usuário necessita que os dados sejam reprocessados e qual é o tempo mínimo para que isso ocorra. Para isso, é possível configurar tarefas por meio do Oozie para que os dados sejam processados com a frequência necessária. O que pode ser em questão de horas ou dias, dependendo da demanda de atualização e do tempo de processar os dados. Em alguns casos, os usuários podem necessitar de dados em tempo real, ou próximo ao tempo real. Para esses casos é recomendável utilizar soluções de *streaming* como Spark Streaming, Storm, Flume ou uma combinação entre Flink e Kafka.

**FINALIZANDO**

Nesta aula, exploramos alguns temas complementares. Diversos outros temas poderiam ser explorados, uma vez que *big data* é uma área muito grande e possui diversas intersecções com várias outras áreas da computação.

No primeiro tema, vimos outros módulos que não foram abordados com tantos detalhes durante as outras aulas. Entre eles, as tecnologias de bancos SQL Impala e Accumulo, as plataformas de processamento e armazenamento em memória Redis e Ignite, o sistema de processamento e distribuição de dados através de grafos NiFi, e a interface web Ambari.

No segundo tema, discutimos as vantagens da abordagem de armazenamento de dados *data lake*, a necessidade de realizar uma governança dos dados através de seus metadados e os seus níveis de maturidade.

No terceiro tema, aprofundamos a discussão sobre os sistemas de recomendação. Discutimos os principais conceitos e motivações por trás dessa tecnologia e os principais tipos de recomendadores. Além disso, analisamos duas das principais abordagens de recomendação individualizada.

No quarto tema, conhecemos os principais conceitos a respeito da computação em nuvem (*cloud computing*), detalhamos os principais modelos de *cloud computing*, entendemos suas relações com a ideia de *big data* *as a service* e conhecemos as principais plataformas de *big data* oferecidas em *cloud computing.*

Por fim, no quinto tema, revisamos os detalhes que devem ser levados em consideração no design de um projeto de *big data*, bem como os principais requisitos que devem ser elencados.

**REFERÊNCIAS**

CHESSELL, N. et al. **Governing and managing Big Data for analytics and decision makers**. New York: IBM Redguides for Business Leaders, 2014.

DIXON, J. Pentaho, Hadoop, and Data Lakes. **James Dixon’s Blog**, 2010. Disponível em: <https://jamesdixon.wordpress.com/2010/10/14/pentaho-hadoop-and-data-lakes/>. Acesso em: 26 nov. 2020.

GORELIK, A. **The enterprise Big Data lake:**delivering the promise of Big Data and Data Science. Sebastopol CA: O'Reilly Media inc., 2019.

LESKOVEC, J; RAJARAMAN, A; ULLMAN, J. **Mining of massive datasets**, Cambridge UK: Cambridge University Press, 2010.