BIG DATA

AULA 5

Prof. Luis Henrique Alves Lourenço

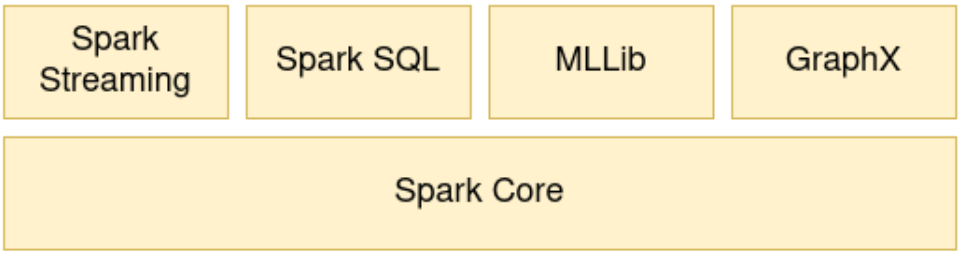
CONVERSA INICIAL

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

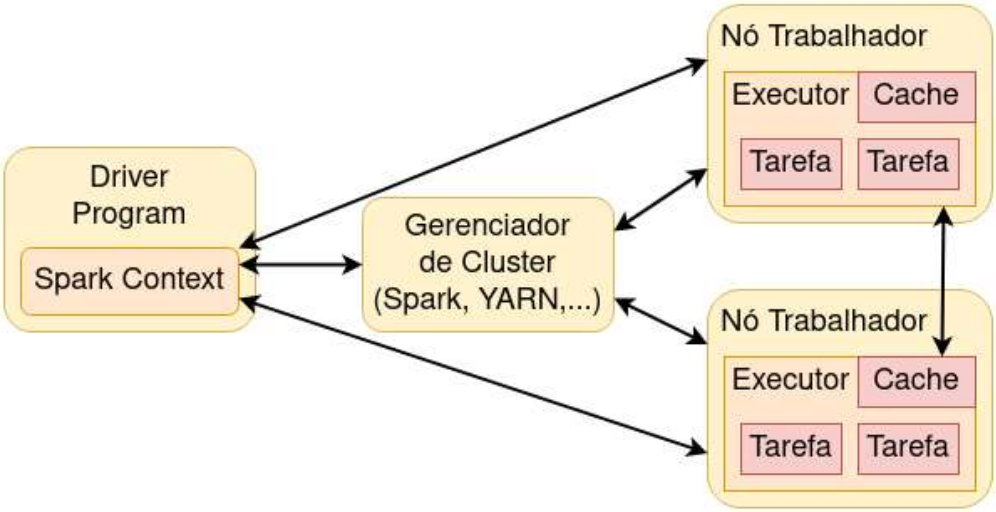


Como podemos ver na Figura 1, os componentes nativos do Spark são Spark *streaming*, Spark SQL, MLLib e GraphX. O Spark *streaming* foi projetado para realizar a ingestão e o processamento de dados em tempo real. O Spark SQL implementa uma interface SQL para o Spark de forma a permitir a utilização de 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*

```
conf = SparkConf().setAppName(appName).setMaster(master)
sc = SparkContext(conf=conf)
```

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

```
distFile = sc.textFile("hdfs://caminho/para/o/dado.txt")
```

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

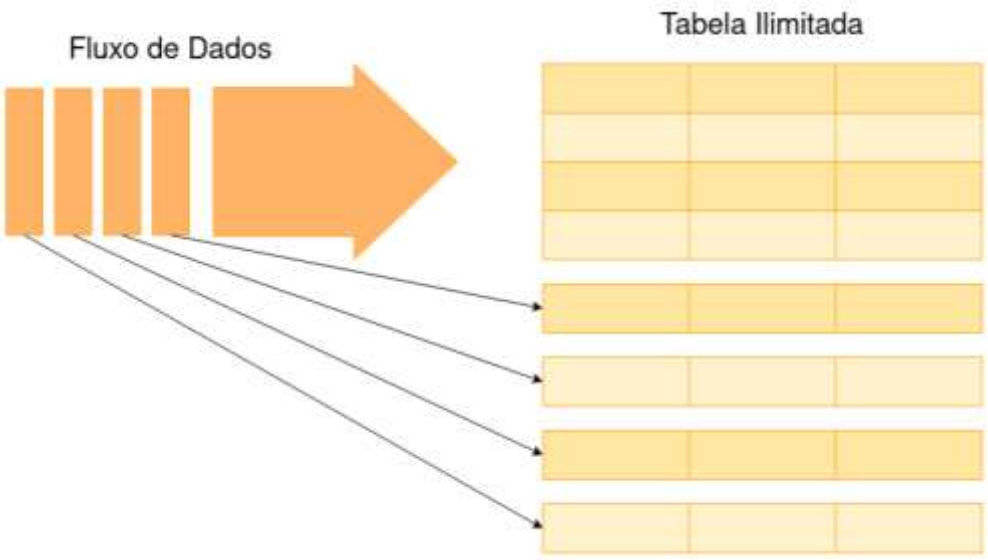
Para submeter aplicações ao *cluster*, há um *script* no diretório *bin* do Spark chamado *spark-submit*, que pode ser utilizado para submeter aplicações em qualquer os gerenciadores de *cluster* suportados pelo Spark. Se a aplicação a ser submetida ao *cluster* depende de outros projetos, deve-se empacotá-las 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



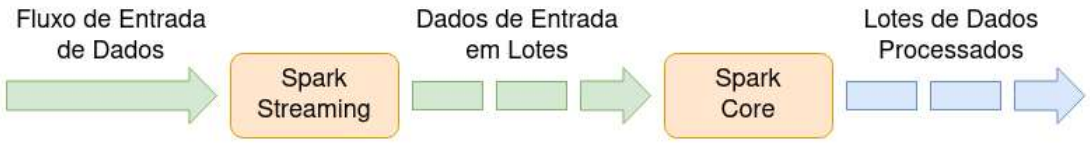
Fonte: Spark SQL, S.d.

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*.

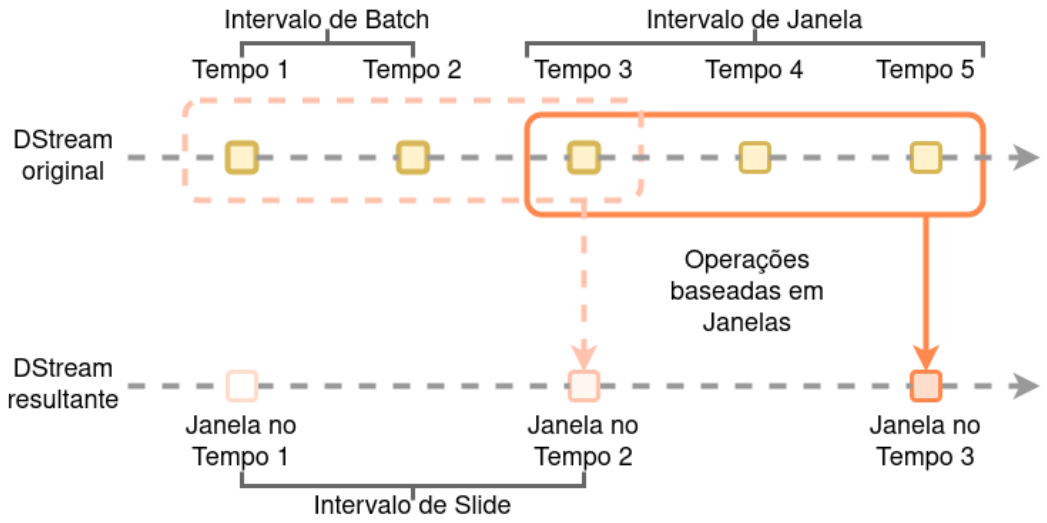
3.1 Fluxos discretizados (*DStreams*)

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*:

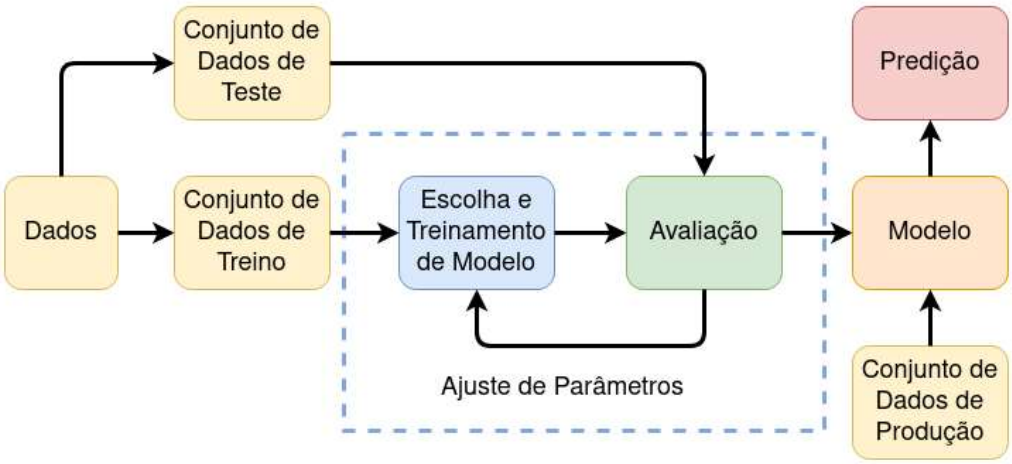
- **Aprendizado supervisionado:** utiliza conjuntos de dados que contêm ambos os dados de entrada e os resultados esperados, ou seja, os dados de entrada são rotulados. Essa categoria pode ser ainda subdividida em dois conjuntos de algoritmos: *classificação* e *regressão*;
- **Aprendizado não supervisionado:**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

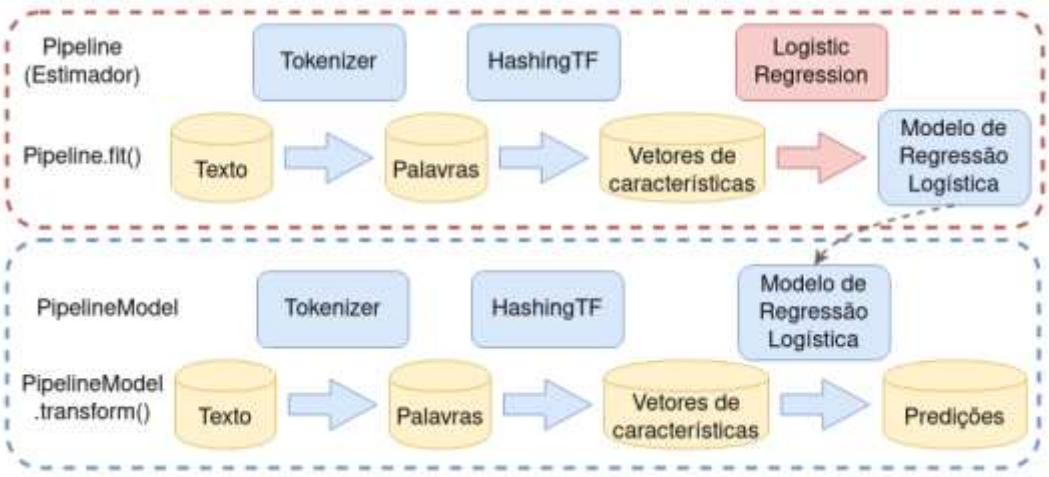
Um estimador abstrai o conceito de um algoritmo de aprendizado, ou qualquer algoritmo de aprendizado que treina sobre um determinado conjunto de dados. É nessa etapa em que modelos são produzidos. O algoritmo de

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.

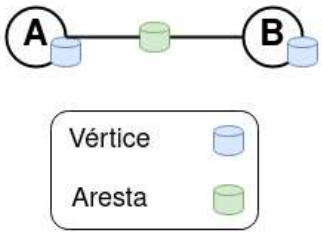
TEMA 5 – GRAPHX

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

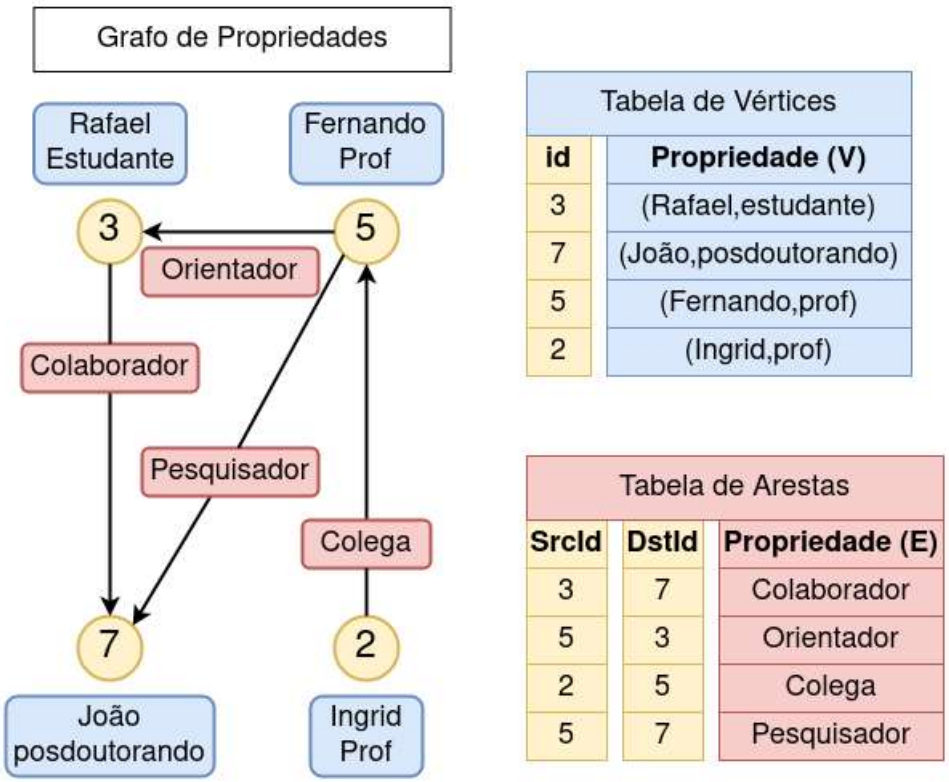
A API GraphX oferece uma forma de armazenar grafos na forma de tabelas e utilizar operações de tabelas para expressar operações de grafos. Assim, facilitando o armazenamento de dados que são mais bem expressos na forma de grafos, e permitindo operar sobre eles da mesma forma como é realizado com qualquer 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

```
// Contabiliza o número de profssores
graph.vertices.filter { case (id, (name, pos)) => pos ==
"prof" }.count
// Contabiliza as arestas onde src > dst
graph.edges.filter(e => e.srcId > e.dstId).count
```

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.