XPe

Aprenda com quem faz

Desenvolvimento de Soluções com Spark_

Pedro Calais

2022

SUMÁRIO

> Capítulo 1

Capítulo 1. Introdução ao Apache Spark

Bem-vindo à apostila sobre Apache Spark! Este material vai te ajudar a se familiarizar com conceitos e paradigmas que vão lhe permitir construir aplicações que lidam com quantidades massivas de dados. Além disso, você vai exercitar estes conceitos em uma das mais relevantes ferramentas de processamento de *big data* da indústria, o que vai lhe capacitar a se desenvolver profissionalmente para atuar como Cientista e Engenheiro de Dados.

Este material está organizado em oito capítulos. Inicialmente, vamos contextualizar o cenário que motiva e demanda a criação de ferramentas e aplicações que processem grandes volumes de dados, além de apresentar em linhas gerais o que é o Spark e porque ele é uma ferramenta útil neste contexto. No capítulo 2, vamos mergulhar na arquitetura do Spark e entender como ele funciona; você vai instalar o Spark e executar algumas aplicações simples. Em seguida, nosso objetivo será habilitá-lo a utilizar o Spark no capítulo 3, para computar estatísticas descritivas sobre grandes volumes de dados – uma competência que todo Cientista de Dados deve dominar. Nos capítulos 4, 5 e 6, você vai conhecer componentes do Spark que lidam com tipos e problemas específicos de dados, respectivamente, dados estruturados, dados em grafos e dados que suportam a criação de modelos de inteligência artificial. No capítulo 7, você vai conhecer o componente do Spark que lida com dados produzidos em fluxos (e potencialmente em tempo real). Finalmente, no capítulo 8, conversaremos um pouco sobre os desafios de se implantar uma aplicação em Spark em produção, “no mundo real”. Ao final da leitura da apostila, você conhecerá um panorama geral e completo sobre as possibilidades que ferramentas como o Spark abrem, visando a extrair valor e conhecimento de grandes volumes de dados. Boa leitura!

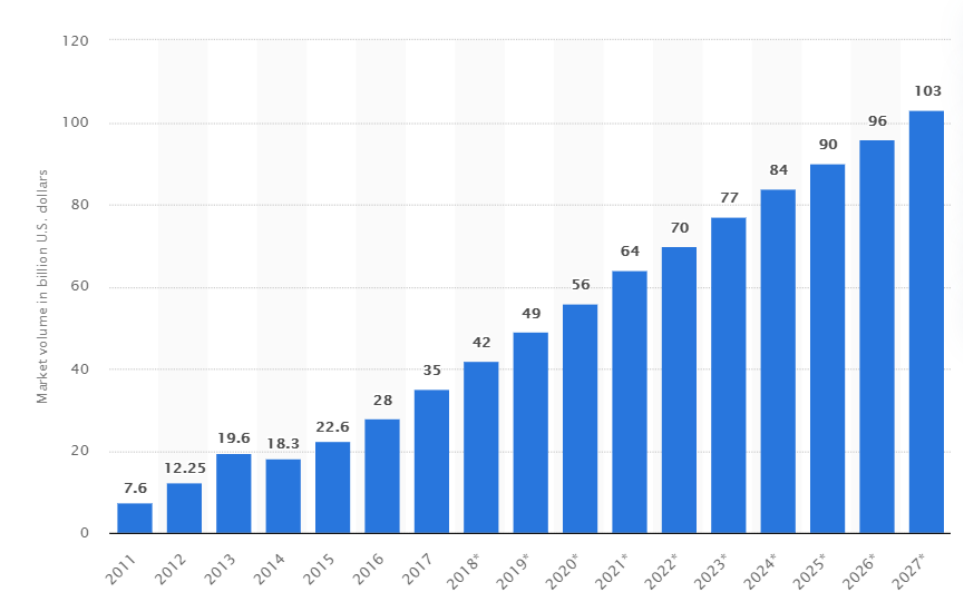
Big Data e o Apache Spark

Neste capítulo, vamos entender como a sociedade cada vez mais digital em que vivemos produz um volume cada vez maior de dados, e como este fenômeno tem imposto desafios tecnológicos que têm impulsionado o desenvolvimento e amadurecimento de ferramentas de processamento de dados em larga escala com o Apache Spark.

Em seguida, vamos definir o Apache Spark em termos das suas principais características, seu histórico, vantagens e desvantagens. Após conhecermos alguns estudos de caso reais em que o Apache Spark foi empregado com sucesso, vamos terminar o capítulo comparando-o com o *Hadoop* e o *MapReduce*, duas tecnologias que o aluno deve saber comparar e relacionar com o Spark.

Big data é o termo usado para se referir a conjuntos de dados que são grandes ou complexos demais para serem processados por aplicações tradicionais, como bancos de dados relacionais. Diversas empresas que oferecem soluções para *big data* têm surgido, e uma pesquisa da *Statista*, empresa líder em dados de mercado e consumidores, estimou em mais de 100 bilhões de dólares anuais a receita de empresas ligadas ao *big data*. Veja a Figura 1.

Figura 1 – A Receita de empresas de *big data* entre 2011 e 2027, em bilhões de dólares, cresce a cada ano.



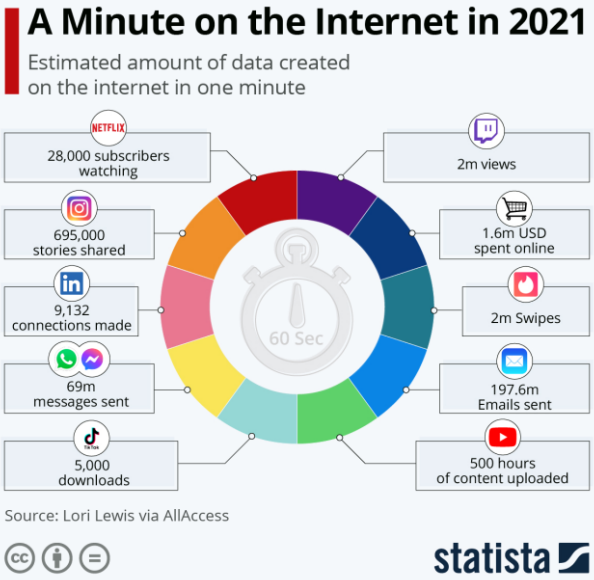
Fonte: statista.com

O infográfico na Figura 2 ajuda a entender o motivo: o volume de dados consumidos e produzidos pela sociedade altamente digital em que vivemos não para de crescer. Em apenas 1 minuto, a quantidade de dados produzida pelas principais empresas ligadas à Internet impressiona:

- Mais de 500 horas de vídeo são produzidas no YouTube.
- Quase 70 milhões de mensagens são trocadas por meio do WhatsApp e Facebook.
- O número de conexões estabelecidas na rede social profissional LinkedIn chega a perto de 10 mil.
- Quase 700 mil *stories* são compartilhados no Instagram.

Ao olharmos de forma mais ampla, a quantidade de dados produzida em um dia deve chegar perto de 460 exabytes em 2025, segundo a consultoria *Recounter*. Um exabyte são 1 bilhões de gigabytes, uma grandeza difícil até mesmo de imaginarmos.

Figura 2 – O que acontece em minuto na Internet?



Fonte: statista.com.

Esses dados e previsões apontam para uma tendência improvável de ser revertida no curto e médio prazo: a necessidade de ferramentas que sejam capazes de extrair valor dos dados segue crescendo. Os benefícios são vários e já foram largamente demonstrados na literatura técnica e de negócios: dados ajudam organizações privadas e públicas a tomarem melhores decisões, reduzirem custos e, em última análise, oferecerem melhores produtos e serviços para seus clientes, usuários e cidadãos.

Ao mesmo tempo em que a sociedade digital produz mais dados, o poder computacional dos computadores e servidores modernos segue crescendo, mas há uma diferença importante aqui: devido às restrições impostas pela lei da física, é cada vez mais difícil tornar computadores mais velozes a um custo economicamente viável. Em particular, existe um limite com o qual um elétron pode se deslocar na CPU de um servidor. Computadores concebidos a partir de uma arquitetura de *hardware* completamente diferente das atuais têm sido exploradas, como arquiteturas baseadas em computação quântica, mas, até elas amadurecem, temos que lidar com um desafio: com o volume de dados cresce a uma taxa maior que

a capacidade de processamento de um computador individual, é cada vez mais comum que cientistas e engenheiros de dados lidem com dados que não cabem no disco ou na memória de uma única máquina, ou ainda, que os dados sejam produzidos a uma taxa tão alta que uma única máquina não disponha de capacidade de processamento para interpretá-los e reagir a eles em tempo hábil.

Estabelecendo isso, o caminho que cientistas e engenheiros de computação tem amadurecido e evoluído ao longo das últimas décadas é: dado que um conjunto de dados é muito grande para ser processado por um nó (ou servidor) em isolado, podemos particioná-lo em vários conjuntos menores e processá-lo de maneira **distribuída**. Assim, múltiplos servidores que podem até mesmo serem máquinas *commodity* de poder computacional similar ao notebook que você provavelmente usa no seu dia a dia, podem ser usados para, de forma paralela e distribuída, processarem e tratarem dados em uma escala muito maior do que uma única máquina poderia fazê-lo. O Google levou essa ideia a uma escala altíssima: é estimado que seus *datacenters* possuam mais de 2,5 milhões de servidores, que em conjunto permitem que seus produtos processem uma massiva quantidade de dados coletados a partir da Web.

Onde entra o Apache Spark nessa história? Bem, sistemas distribuídos são difíceis de desenhar e construir: é preciso lidar com falhas dos servidores e cuidadosamente implementar a divisão e coordenação de tarefas entre as máquinas independentes. Por isso, várias ferramentas que abstraem do programador a necessidade de conhecer esses detalhes foram projetadas, tais como o Hadoop/Map Reduce, Apache Flink, Apache Storm e o próprio Apache Spark, nosso foco neste curso.

Ao pesquisarmos em uma máquina de busca como o Google por definições acerca do Spark, pelo menos uma dezena delas surge a partir dos

principais livros, *tutoriais* e publicações sobre a plataforma. Vamos listar as mais frequentes:

Segundo o Google, o Spark é:

- Uma plataforma para big data.
- Um framework para processamento de dados.
- Uma ferramenta de propósito geral para processamento distribuído de dados.
- Uma tecnologia para computação em cluster ultra-rápida.
- Uma ferramenta para análise de dados e machine learning unificada e ultra-rápida.
- Um sistema em código-aberto para processamento distribuído usado para cargas de trabalho de big data.

Os principais conceitos do Spark se repetem nessas definições:

- O foco do Spark é processar grandes volumes de dados -- *big data*.
- O Spark tem capacidade para desempenhar este papel de forma distribuída, isto é, dividindo a carga de trabalho em servidores independentes. O Spark abstrai do programador a necessidade de coordenar e programar a divisão da tarefa de dados nas máquinas independentes; o sistema cuida desta missão provendo *paralelismo implícito de dados* e *tolerância a falhas*.
- O Spark é uma plataforma unificada: ele consegue lidar com diferentes cargas de trabalho: dados em lote (*batch*), em fluxo (*streaming*), dados estruturados e dados que vão suportar a criação de modelos de aprendizado de máquina.

- O Spark é projetado pensando em desempenho de execução como característica fundamental.

- O Spark é um sistema em código aberto.

O Spark foi criado no laboratório de pesquisa AMPLab, na Universidade da Califórnia, Berkeley, em 2009. Quatro anos depois foi incorporado à Fundação Apache e seus criadores fundaram a Databricks, empresa que explora comercialmente o Spark por meio de treinamentos, consultorias e produtos que "empacotam" o Spark junto com outras soluções de big data e plataformas de gestão de clusters. Desde então, dezenas de livros técnicos sobre a ferramenta foram escritos (busque por "Apache Spark" na Amazon) e meetups foram formados. Hoje, mais de 1.000 contribuidores oriundos de uma centena de organizações já ajudaram a escrever parte do código do Spark, disponível em: https://github.com/apache/spark. Ao longo dos próximos capítulos vamos mergulhar em códigos de aplicação Spark, mas caso você esteja curioso e queira ver desde já como é um código típico de uma aplicação Spark, há vários deles no site oficial: https://spark.apache.org/docs/latest/quick-start.html. Até o fim do curso você entenderá em um nível suficiente todas as linhas do programa de exemplo deste link!

Vantagens e desvantagens do Spark

Três características tornaram o Apache Spark uma ferramenta importante e relevante no ecossistema de *big data*: velocidade, simplicidade de uso e o fato de que ela é uma plataforma unificada. Vamos explorar cada uma dessas características e, ao final, discutir em que casos o Spark provavelmente não se enquadra como uma boa ferramenta.

A primeira característica importante do Apache Spark é a **velocidade** ao processar grandes massas de dados. Dois princípios de desenho cumprem um papel importante aqui. O primeiro é que a arquitetura do Spark (que vamos entender em detalhes no Capítulo 2) é bastante alinhada com a

forma como a indústria de hardware evoluiu: servidores baratos, com muita memória e *cores* de processamento. Em segundo lugar, o Spark foi desenhado para privilegiar o máximo possível de processamento em memória, em contrapartida ao Hadoop, cujas aplicações tipicamente executa mais leituras e escritas de dados em disco.

Ainda em 2014, o Apache Spark foi avaliado contra um benchmark de ordenação, cuja entrada variava entre 100 TB e 1000 TB. Os resultados, na Figura 3, mostram o poder computacional que o Spark consegue destravar: 100 TB foram ordenados em apenas 23 minutos, por meio de 206 nós que continham 6.592 *cores*.

Figura 3 – Benchmark de ordenação de dados: aplicação em Hadoop MapReduce para ordenar 100 TB, em Spark para ordenar 100 TB e em Spark para ordenar 1 PB. O Spark venceu o Hadoop MapReduce em desempenho.

		Spark 100 TB *	Spark 1 PB
Data Size	102.5 TB	100 TB	1000 TB
Elapsed Time	72 mins	23 mins	234 mins
# Nodes	2100	206	190
# Cores	50400	6592	6080
# Reducers	10,000	29,000	250,000
Rate	1.42 TB/min	4.27 TB/min	4.27 TB/min
Rate/node	0.67 GB/min	20.7 GB/min	22.5 GB/min
Sort Benchmark Daytona Rules	Yes	Yes	No
Environment	dedicated data center	EC2 (i2.8xlarge)	EC2 (i2.8xlarge)

Fonte: databricks.com.

A segunda característica importante do Apache Spark é o esforço que seus contribuidores colocaram em torná-la uma ferramenta **simples** de ser usada pelos programadores. No Capítulo 2 mostraremos detalhes, mas, em essência, você já pode saber que uma aplicação Spark é construída a

partir de um conjunto de operações que executam sobre uma estrutura de dados chamada RDD - *Resilient Distributed Dataset* - que retira do programador a necessidade de conhecer os detalhes de como distribuir a computação entre servidores. A Figura 4 ilustra a simplicidade do Spark: uma aplicação que conta palavras de um arquivo potencialmente grande (e que pode estar armazenado em um sistema de arquivos distribuído como o HDFS) tem apenas cinco linhas.

Figura 4 – Contando palavras em uma aplicação Spark.

```
val textFile = sparkSession.sparkContext.textFile("hdfs:///tmp/words")
val counts = textFile.flatMap(line => line.split(" "))
                     .map(word => (word, 1))
                     .reduceByKey(_ + _)
counts.saveAsTextFile("hdfs:///tmp/words_agg")
```

Fonte: databricks.com.

Ainda na linha de permitir que os programadores sejam produtivos, o Spark, embora implementado na linguagem de programação Scala, suporta, além do Java, Python e R, o que o torna especialmente conveniente para ser adotado por cientistas de dados, que usam intensamente bibliotecas de aprendizado de máquina e análise de dados dessas linguagens.

Finalmente, vale destacar uma característica importante do Spark: o fato de que ele é uma plataforma **unificada**. O Spark consegue lidar com diversas cargas de trabalho: dados em lote, em streams, dados que vão produzir modelos de aprendizado de máquina, dados estruturados, dados em grafos. O suporte a essa diversidade de cargas de trabalho é especialmente útil, pois evita que o Cientista de Dados precise lidar – e aprender – várias ferramentas distintas, uma para cada tipo de dado. Por exemplo, o Cientista de Dados poderia adotar o Hadoop para processar dados em lote, o Google Pregel para processar grafos, e o Apache Storm para processar fluxos de dados. Note que são várias APIs para aprender e sistemas para configurar. Por isso, o fato do Spark reunir em uma única

plataforma a capacidade de lidar com diversas cargas de trabalho é bastante valorizada por profissionais de Ciência de Dados.

Há pelo menos um cenário em que o Spark pode não ser a melhor escolha: a arquitetura do Spark foi projetada tendo-se em mente abundância de memória e recursos computacionais, se os nodos do *cluster* tiverem pouca memória, provavelmente fará mais sentido usar o Hadoop MapReduce, que, por sua vez, usa mais o disco do que o Spark. Além disso, a complexidade de se configurar e implantar o Spark pode não se justificar caso você não tenha a necessidade de processar vários gigabytes de dados de forma rápida. Caso o volume de dados que você tenha em mãos seja relativamente pequeno, escrever programas em Python utilizando bibliotecas como o *Pandas* pode ser mais produtivo para você.

Estudos de caso reais

Nesta seção, vamos nos dedicar a ilustrar, com casos reais, a aplicabilidade do Apache Spark em lidar com problemas de dados, a fim de motivá-lo a perceber o potencial da ferramenta em ajudá-lo a resolver seus próprios problemas ligados à dados por meio do Spark.

Tipicamente, dois perfis de programadores são o público-alvo do Spark: Cientistas de Dados e Engenheiros de Dados.

Ciência de dados é uma área que combina matemática, estatística, ciência da computação e programação. Cientistas de dados se concentram em "contar estórias" por meio dos dados, descobrir padrões e criar modelos de previsão e recomendação a partir dos dados (depois de gastar um tempo significativo limpando os dados, claro). Componentes do Spark como o Spark MLLib (para aprendizado de máquina) e Spark SQL (para exploração interativa e *ad hoc* dos dados) são potencialmente úteis para este público.

Engenheiros de Dados, por sua vez, estão mais preocupados em implantar modelos em produção, a transformar dados crus e sujos em dados

úteis, e em criar um *pipeline* de dados fim-a-fim, conectando-se com fontes e destinos de dados como aplicações Web e sistemas de armazenamento de dados como bancos SQL e NoSQL. Para este público, o Spark tende a ser útil pois fornece uma API simples que esconde do programador a complexidade de distribuir os dados entre servidores e de tolerar falhas, além de prover APIs para ler e combinar dados oriundos de múltiplas fontes.

Em resumo, podemos listar os seguintes casos de uso como bastante típicos do mundo Apache Spark:

- Processar em paralelo grandes conjunto de dados.
- Explorar e visualizar dados e forma interativa.
- Construir, treinar e avaliar modelos de aprendizado de máquina.
- Analisar redes sociais e grafos em geral.
- Implementar *pipelines* de dados fim a fim a partir de fluxos de dados.

Vamos ver alguns exemplos reais de uso do Spark.

WorldSense

A WorldSense foi uma *startup* de publicidade on-line sediada na cidade de Belo Horizonte, que operou entre 2015 e 2018. O desafio tecnológico demandava a indexação de um conjunto de documentos da Web bastante grande, conhecido como *Common Crawl* - https://commoncrawl.org/.

O conjunto de dados que a empresa precisava processar era bastante grande:

- 2 bilhões de documentos da Web.
- 40 bilhões de *links* entre os documentos.

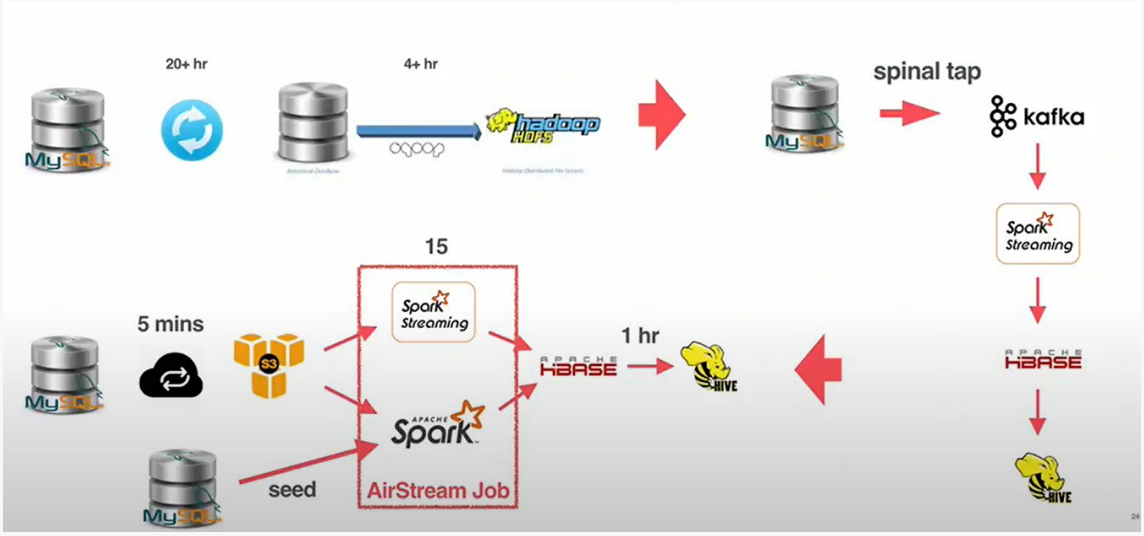
- Cerca de 40 TB de dados!

Cada documento era alvo de manipulação computacionalmente intensiva, como realizar *parsing* do HTML e extrair os *links*. O processamento sequencial deste conjunto de dados demoraria semanas; a partir do Apache Spark e utilizando um cluster de máquinas na Amazon AWS, a empresa conseguia processar os dados em cerca de algumas horas, o que foi importante para a empresa conseguir iterar e evoluir seu modelo de negócio com velocidade.

Airbnb

O Airbnb, um serviço que conecta pessoas interessadas em aluguéis de imóveis, possui um *pipeline* de dados complexo e precisa ler e gravar em fontes de dados distintas. A empresa usa o Apache Spark, inclusive as funcionalidades de *streaming*, como ilustrado na Figura 5. O sistema da empresa processa dados no Spark tanto em *batch* quanto em *streaming*, o que concretamente ilustra uma das vantagens que apontamos sobre o Spark – o fato de ele ser uma plataforma unificada.

Figura 5 – A arquitetura de dados do Airbnb usa o Apache Spark.

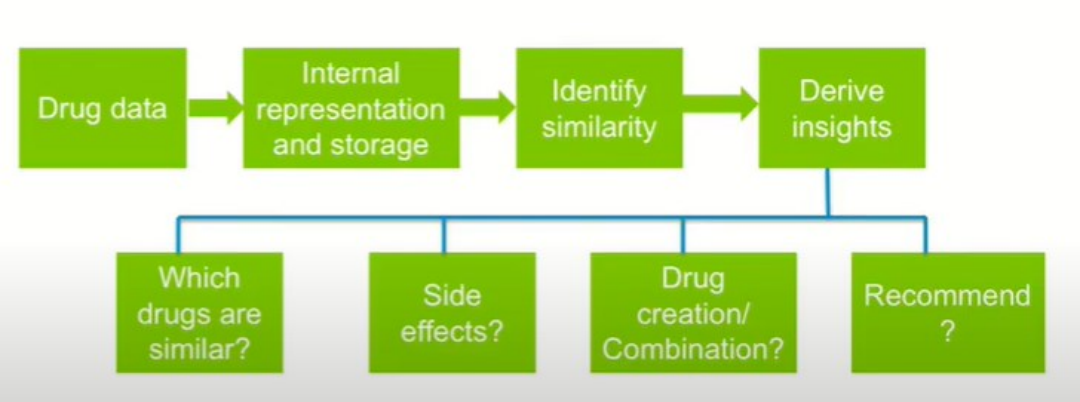


Fonte: databricks.com.

Intel

O grupo de inteligência artificial e *analytics* da Intel utiliza o Apache Spark no domínio da bioinformática. A partir de dados das estruturas de moléculas, que são naturalmente representadas como grafos, o grupo construiu em Python um sistema que, apoiado pelas APIs do Spark, consegue extrair características das drogas e compará-las, visando responder perguntas como estimar possíveis efeitos colaterais de novos remédios. A Figura 6 ilustra o fluxo de processamento que usa o Spark.

Figura 6 – Fluxo de processamento de dados sobre medicamentos usados pela Intel e que emprega o Apache Spark como uma das ferramentas.



Fonte: databricks.com

NBC

A NBC é um dos maiores grupos de mídia do mundo, e precisa servir em tempo real vídeos sob demanda para os consumidores de sua plataforma. O problema de negócio é bastante claro: se todos os vídeos foram mantidos em *cache*, a latência será baixa e os usuários experimentarão uma ótima experiência ao visualizar os vídeos; por outro lado, esta solução, no entanto, é cara. Por outro lado, se nenhum vídeo for mantido em *cache*, os usuários experimentarão lentidão. A decisão a ser tomada aqui é quando e quais vídeos devem ser trazidos para o *cache*. A NBC criou um *pipeline* de dados que, toda madrugada, lê os metadados dos vídeos (como tamanho, duração e id), cria visões agregadas e guarda em um banco HBase. A caracterização gerada permite decidir com assertividade

quais vídeos devem ser cacheados para otimizar a relação custo-benefício para a NBC e seus usuários.

Apache Spark versus Apache Hadoop MapReduce

No ecossistema de plataformas para *big data*, o Apache Spark tem um importante "concorrente": o Apache Hadoop MapReduce, que é o módulo de processamento do Hadoop. O Hadoop nasceu em 2006 e a descrição oficial o define como "uma plataforma para escrita de aplicações que processam grandes volumes de dados, em paralelo, em grandes *clusters* de máquinas *commodity*, de maneira confiável e tolerante a falhas". Assim como Spark, o Hadoop é uma plataforma composta por vários módulos:

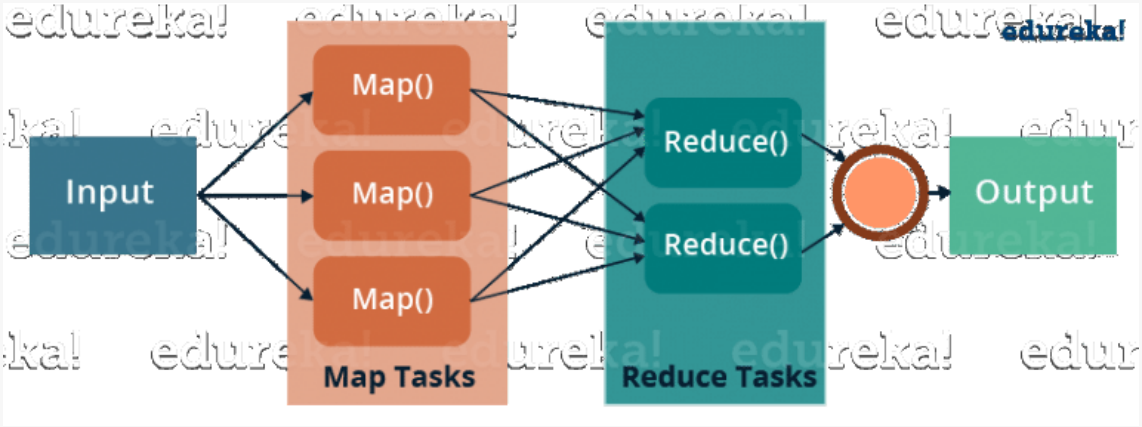
- Hadoop Common.
- Hadoop Distributed File System (HDFS), um sistema de arquivos distribuído.
- Hadoop YARN, um gestor de cluster.
- Hadoop Ozone, um banco de dados de objetos.
- Hadoop MapReduce, o motor de processamento que implementa o modelo de programação MapReduce.

Uma confusão costuma acontecer em torno do que significa o Hadoop: ora a comunidade quer mencionar toda a plataforma, ora apenas o motor de processamento MapReduce. Vamos clarificar: o Spark e o Hadoop podem cooperar no sentido de que uma aplicação Spark pode usar o HDFS como sistema de arquivos distribuído, isto é, podemos usar o sistema de processamento do Spark para ler e processar arquivos gigantescos armazenados no HDFS. Também podemos usar o gestor do cluster do Hadoop, o YARN - *Yet Another Resource Navigator* – como gestor do cluster com o qual o Spark vai interagir.

A comparação que faz sentido, portanto, é entre o motor de processamento de dados do Spark e do Hadoop (MapReduce). O MapReduce é um modelo de programação paralelo em que a computação acontece em duas etapas: uma etapa de mapeamento e uma etapa de redução. Veja a Figura 7.

- *map:* filtra e ordena os dados aos expô-los como pares (chave, valor).
- *reduce*: a partir do mapeamento anterior, sumariza os dados e produz o resultado final.

Figura 7 – O Modelo de programação *MapReduce*.

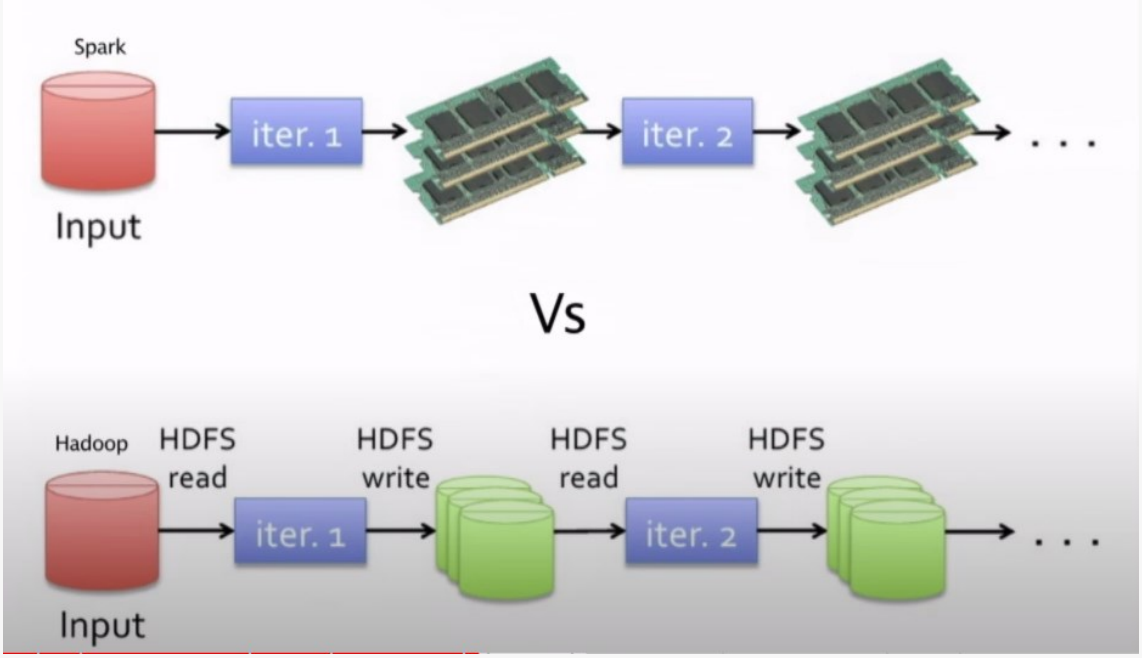


Fonte: edureka.com

Existe uma grande diferença de implementação entre o Spark e o Hadoop MapReduce: entre as iterações de mapeamento e redução, o Hadoop grava e lê os dados em disco; enquanto o Spark mantém os dados em memória. Essa diferença impacta muito no desempenho: aplicações Spark podem ser entre 10 e 100 vezes mais velozes que aplicações equivalentes no Hadoop, por executarem menos leituras e gravações em disco. Em particular, o Spark tende a brilhar em tarefas iterativas, em que o mesmo dado é lido várias vezes (pense, por exemplo, em algoritmos clássicos de aprendizado de máquina que iteram sobre os dados até a convergência, como descendente gradiente). O Hadoop MapReduce desempenha mal

neste tipo de tarefa, pois a cada iteração o dado será lido novamente do disco. Veja a Figura 8.

Figura 8 – Diferença entre Spark e Hadoop MapReduce. O Spark privilegia o uso da memória, enquanto o Hadoop MapReduce usa mais o disco para armazenar os resultados intermediários das computações.



Fonte: Music Recommendations at Scale with Spark - Christopher Johnson (Spotify).

Em relação à tolerância a falhas, tanto o Spark como o Hadoop MapReduce são desenhados para oferecer esta capacidade; como o Hadoop usa mais o disco, pode ser recuperar mais rápido de falhas, de modo que podemos dizer que ele é um “pouco mais tolerante a falhas”.

Uma diferença importante entre o Hadoop e o Spark é que o Hadoop é projetado para lidar com aplicações em *batch*, isto é, que lê um conjunto de dados e o processa de uma única vez. O Spark, por sua vez, lida com dados em *batch*, *streaming*, que podem estar representados como grafos e suportar a construção de modelos de aprendizado de máquina - de forma unificada, como vimos na Seção 1.2.

Em resumo, componentes do Hadoop como o HDFS (sistema de arquivos distribuído) e YARN (gestor de cluster) podem ser usados em conjunto com o Spark. Sobre o motor de processamento dos dados, o Spark tende a ser mais veloz que o Apache MapReduce, porque o último grava e lê dados em disco a cada operação, enquanto o Spark privilegia manter os dados em memória.

> Capítulo 2

Capítulo 2. Conceitos fundamentais do Apache Spark

Neste capítulo iremos apresentar a arquitetura do Spark, sua estrutura de dados fundamental – o RDD, bem como as principais operações que podem ser executadas sobre RDDs. Vamos também examinar alguns exemplos concretos de aplicações Spark. Você vai aprender a instalar o Spark e vai começar a se familiarizar com o código de aplicações Spark.

A arquitetura do Apache Spark

Já sabemos que o Spark é uma plataforma para processamento de grandes volumes de dados em um ambiente de computação distribuída, isto é, composta por mais uma unidade de computação independente (vários computadores reais ou virtuais). Vamos, agora, entender mais sobre a arquitetura do Spark, seus principais componentes e como eles interagem.

A Figura 9 ilustra a arquitetura geral do Spark. Ela é composta por três componentes principais:

- O driver program.
- O gestor do cluster (cluster manager).
- Um conjunto de worker nodes.

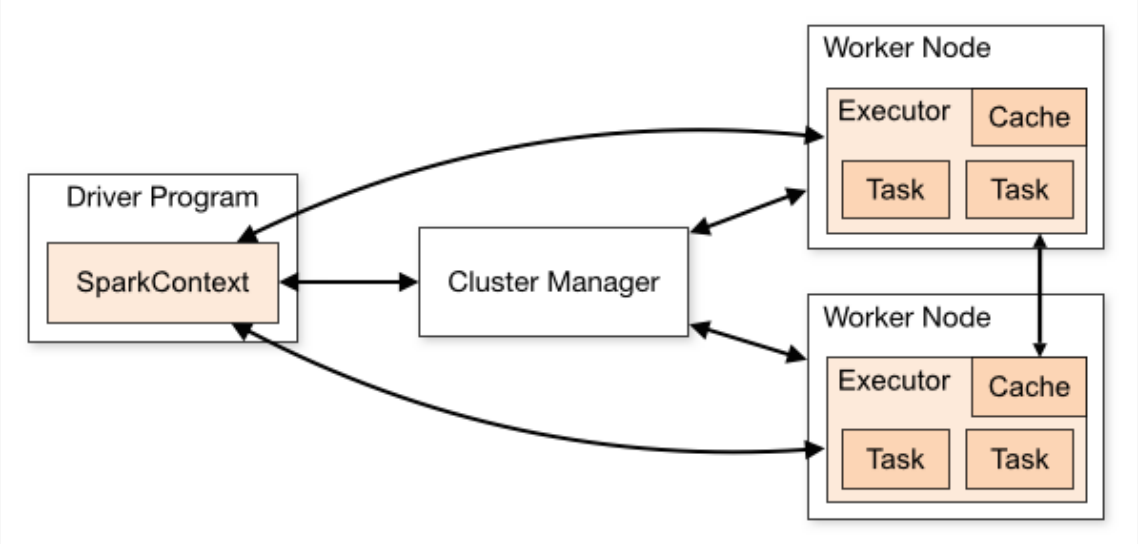
O *driver* é o "main()" de uma aplicação Spark. Ele é responsável por:

- Orquestrar as operações paralelas executadas no cluster.
- Comunicar com o gestor do cluster e agendar operações a serem executadas no cluster.
- Distribuir a alocação de recursos nos executores, que são os processos que de fato executam as tarefas, dentro dos nodos que chamamos de *workers*.

O *cluster manager*, por sua vez, é um serviço que gerencia e aloca recursos do *cluster*. Vários gestores de cluster podem ser "plugados" aqui: o gestor de *cluster* padrão do Spark ou outros, como o Hadoop YARN, Apache Mesos e Kubernetes.

Por fim, os *workers* são nós do cluster que executa tarefas da aplicação Spark. O processo dentro do *worker* que executa a aplicação é chamado de *executor*.

Figura 9 – A arquitetura do Apache Spark.



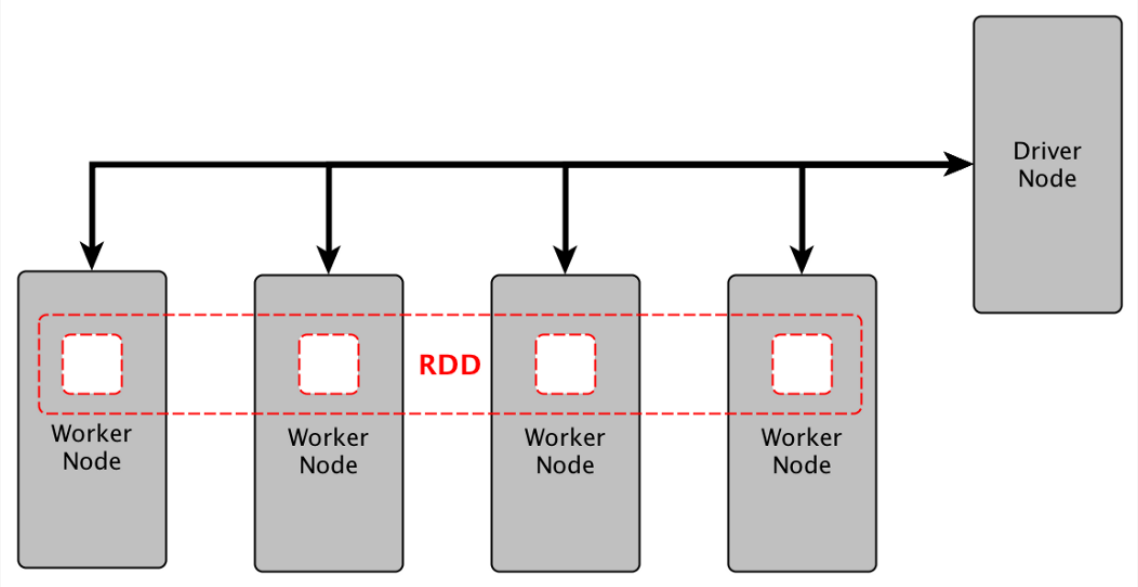
Fonte: https://spark.apache.org/.

RDD

O RDD é a principal estrutura de dados e abstração do Spark. RDD significa *Resilient Distributed Dataset*, (1) um conjunto de dados que é (2) distribuído no cluster e (3) tolerante (resiliente) a falhas. A Figura 10 representa um RDD: o programador interage com uma estrutura de dados que, por debaixo dos panos, representa uma coleção de objetos que está particionada fisicamente no cluster, em potencialmente dezenas ou milhares de nós. O programador, no entanto, não precisa conhecer como isso é feito; ele lida com uma estrutura de dados única e o fato de que os dados estão potencialmente particionados no cluster é transparente para ele. Mais formalmente, o RDD é uma abstração de memória distribuída que permite

aos programadores executarem computações em memória em *clusters* de maneira tolerante a falhas.

Figura 10 – RDDs: *Resilient Distributed Datasets*. O RDD guarda os dados de forma distribuída em vários nodos de forma transparente para o programador.



Fonte: https://medium.com/@lavishj77/spark-fundamentals-part-2-a2d1a78eff73.

O Spark tenta manter, sempre que possível, os RDDs na memória dos nodos do cluster. Diferentemente de outros *frameworks* como o Apache Hadoop MapReduce, manter os dados em memória traz ganhos de desempenho de uma ordem de magnitude ou, mais especialmente, em algoritmos iterativos (típicos do mundo de aprendizado de máquina), quando os mesmos dados são lidos processados múltiplas vezes.

Uma aplicação Spark tipicamente opera sobre RDDs da seguinte forma:

1. Lê dados do disco (potencial, de um sistema de arquivos distribuído como o HDFS) para um RDD.

2. Aplica uma sequência de **transformações** sobre um RDD, como, por exemplo, filtrar o RDD para apenas manter itens que satisfazem um critério.

3. Aplica uma sequência de **ações** que retornam para o *driver* um resultado, por exemplo, contar quantos itens fazem parte do RDD.

Existem três formas de se criar um RDD. A primeira é a partir de uma coleção em memória, o *driver* vai dividir a coleção em partes e enviá-las para os *workers*. A Figura 11 mostra um código em Python que lê uma lista da memória do *driver* e cria um RDD.

Figura 11 – Criando um RDD a partir de uma coleção em memória.

```
In [1]:
from pyspark.sql import SparkSession

In [4]:
spark = SparkSession\
        .builder\
        .appName("PythonNumberCount")\
        .master("local[2]")\
        .getOrCreate()

list = [1, 2, 3, 4, 5]
rdd1 = spark.sparkContext.parallelize(list)

In [5]:
rdd1

Out[5]: ParallelCollectionRDD[0] at readRDDFromFile at PythonRDD.scala:274
```

A segunda forma é a partir da leitura de um arquivo que está no sistema de arquivos local ou um sistema de arquivos distribuído como o Apache HDFS:

Figura 12 – Criando um RDD a partir de um arquivo.

```
In [8]:
rdd2 = spark.sparkContext.textFile("/opt/spark/README.md")
rdd2

Out[8]: /opt/spark/README.md MapPartitionsRDD[4] at textFile at NativeMethodAcce
ssorImpl.java:0
```

Finalmente, a terceira forma é criar um RDD a partir de um RDD existente, a partir de uma operação de **transformação**. Na Figura 13 o *rdd2* é alvo de uma transformação *flatMap*(), que quebra as linhas do arquivo em palavras e produz um *rddf* cujos objetos são palavras do arquivo. No decorrer

deste capítulo vamos estudar em detalhes as principais transformações que podem ser aplicadas sobre RDDs.

O Spark implementa tolerância a falhas da seguinte forma: o sistema mantém registro das transformações executadas sobre um RDD; caso a comunicação com o executor responsável por uma partição do RDD seja perdida, o Spark detém toda a informação necessária para recomputar o estado do RDD desde o princípio. A sequência de transformações aplicadas a um RDD é chamada no mundo Spark de *lineage* (linhagem).

Figura 13 – Criando um RDD a partir de uma transformação.

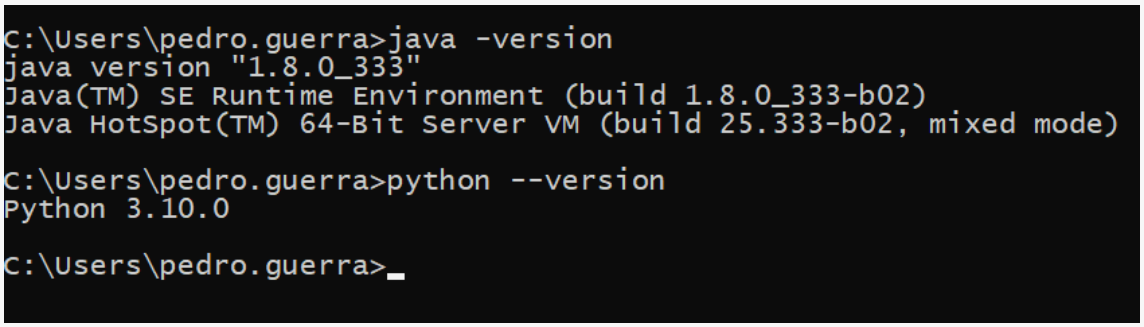
```
In [9]:  1  rddf = rdd2.flatMap(lambda line:line.split())

In [10]: 1  rddf

Out[10]: PythonRDD[5] at RDD at PythonRDD.scala:53
```

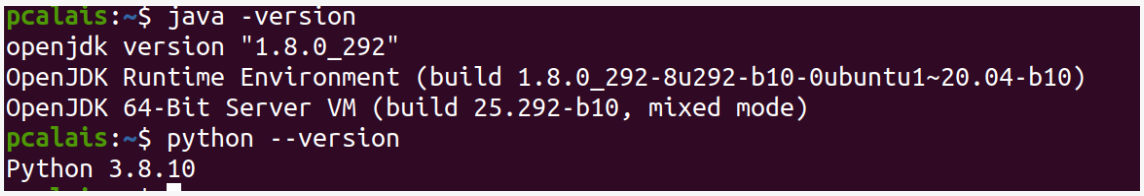
Instalando o Spark

Nesta seção vamos baixar e instalar o Spark. Dois links úteis para instalar o Spark no Linux e no Windows são: https://phoenixnap.com/kb/install-spark-on-ubuntu para o Linux e https://phoenixnap.com/kb/install-spark-on-windows-10 para o Windows. **Siga com atenção o passo a passo de um desses links!**

- Você deve começar confirmando que você tem o Java e o Python instalado na sua máquina, com os comandos **java -version** e **python --version.** Caso contrário, instale o Java e o Python.

Figura 14 – Verificando que o Java e o Python estão instalados. No Windows e no Linux (Primeiro e segundo screenshots, respectivamente), os comandos são os mesmos.



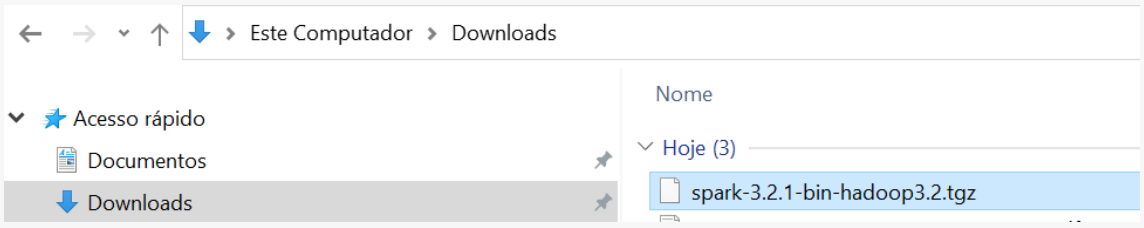


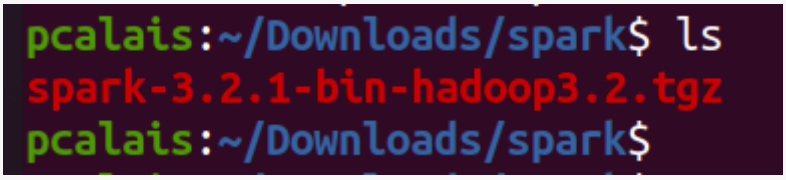
- Baixe o Spark em https://spark.apache.org/downloads.html; no screenshot que você vê na figura 15, o arquivo a ser baixado é *spark-3.2.1-bin.hadoop3.2.tgz.*

Figura 15 – Site para download do Spark.

Confirme que o arquivo foi baixado para a sua máquina:

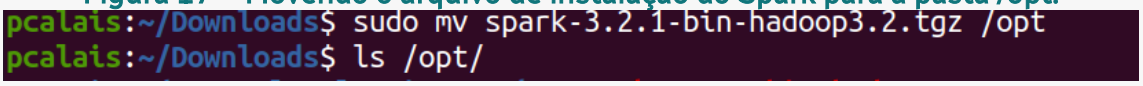
Figura 16 – Verificando que arquivo contendo a instalação do Spark foi baixado no Windows e Linux, respectivamente.





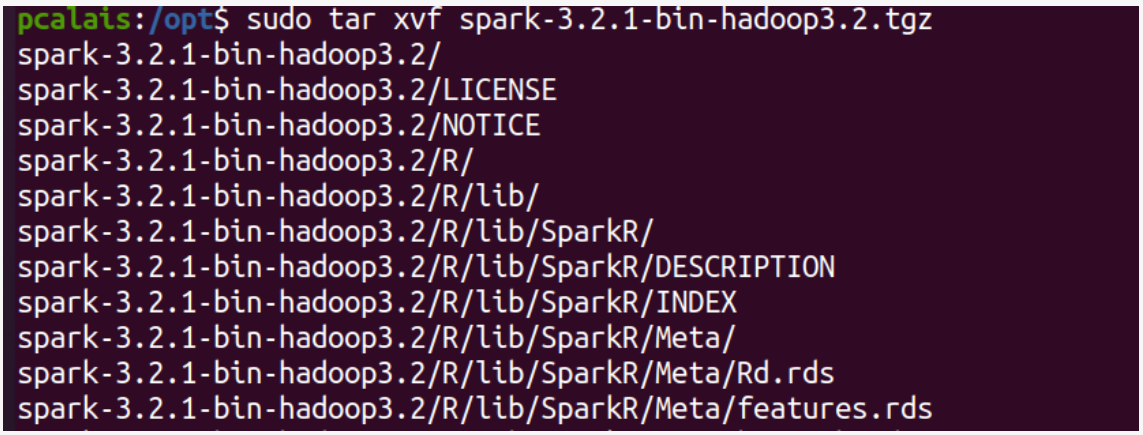
No Linux, mova o arquivo .tgz para a pasta /opt:

Figura 17 – Movendo o arquivo de instalação do Spark para a pasta /opt.



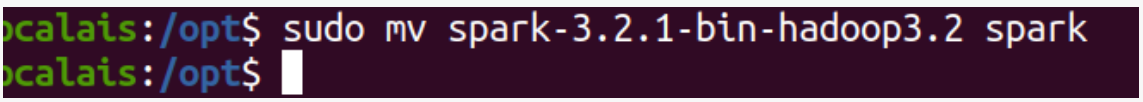
Em seguida, vá para a pasta */opt* (com *cd /opt*) e descompacte o arquivo:

Figura 18 – Descompactando o arquivo de instalação do Spark.



Em seguida, renomeie o diretório recém-criado *spark-3.2.1-bin-hadoop3.2* para *spark*, para facilitar nosso trabalho:

Figura 19 – Renomeando a pasta que contém os arquivos do Spark.



- No Linux, ajuste as variáveis de ambiente como demonstrado no link https://phoenixnap.com/kb/install-spark-on-ubuntu e na Figura 14.

Figura 20 – Ajustando as variáveis de ambiente do Spark no Linux.

```
$ echo "export SPARK_HOME=/opt/spark" >> ~/.profile
$ echo "export PATH=$PATH:$SPARK_HOME/bin:$SPARK_HOME/sbin" >> ~/.profile
$ echo "export PYSPARK_PYTHON=/usr/bin/python3" >> ~/.profile
$ source ~/.profile
$
```

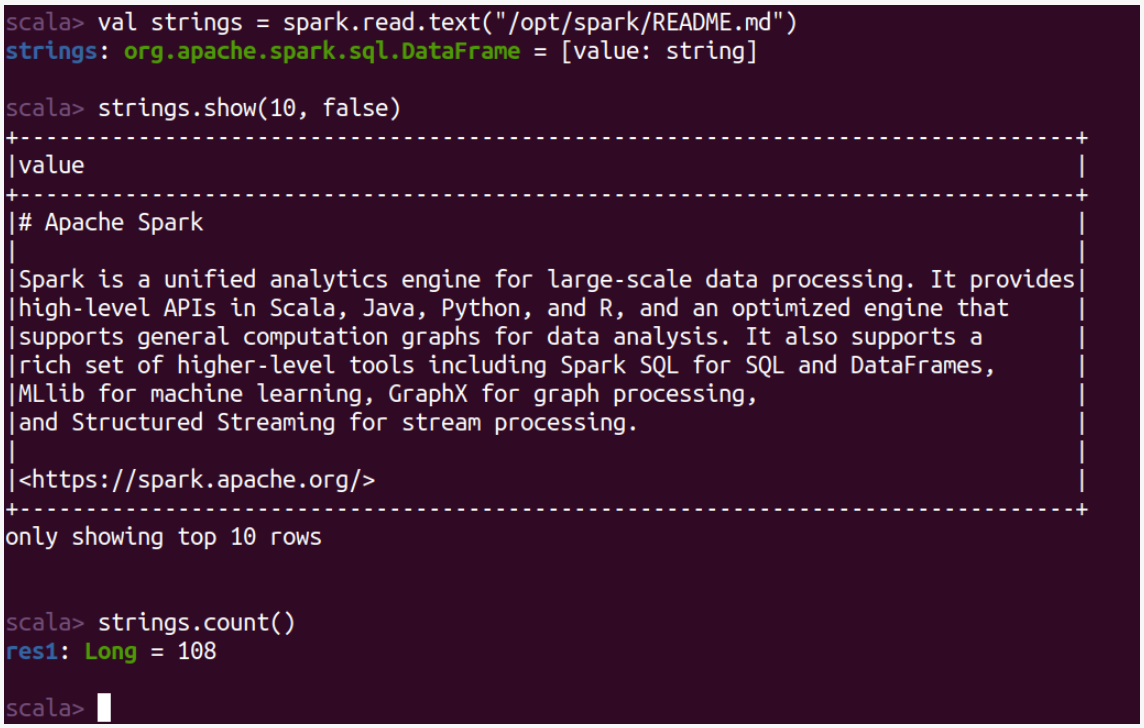
Com o Spark instalado, você pode executar o *spark-shell* e ter acesso a um ambiente interativo em que você pode programar na linguagem Scala e interagir com o Spark:

Figura 21 – Entrando no *spark-shell*.

```
pcalais:~/Downloads$ spark-shell
21/10/17 19:12:20 WARN Utils: Your hostname, pcalais-Inspiron-15-7000-Gaming resolves to a loopback address: 127.0.1.1; using 192.168.0.11 instead (on i
nterface wlp3s0)
21/10/17 19:12:20 WARN Utils: Set SPARK_LOCAL_IP if you need to bind to another address
21/10/17 19:12:21 WARN NativeCodeLoader: Unable to load native-hadoop library for your platform... using builtin-java classes where applicable
Using Spark's default log4j profile: org/apache/spark/log4j-defaults.properties
Setting default log level to "WARN".
To adjust logging level use sc.setLogLevel(newLevel). For SparkR, use setLogLevel(newLevel).
Spark context Web UI available at http://192.168.0.11:4040
Spark context available as 'sc' (master = local[*], app id = local-1634508745785).
Spark session available as 'spark'.
Welcome to
      ____              __
     / __/__  ___ _____/ /__
    _\ \/ _ \/ _ `/ __/  '_/
   /___/ .__/\_,_/_/ /_/\_\   version 3.1.2
      /_/

Using Scala version 2.12.10 (OpenJDK 64-Bit Server VM, Java 1.8.0_292)
Type in expressions to have them evaluated.
Type :help for more information.

scala> spark.version
res0: String = 3.1.2
```

Veja na Figura 21 que o sistema carrega o objeto *spark,* que é do tipo *SparkSession*: o *SparkSession* é um objeto como ponto de entrada para interagir com o Spark. No spark-shell, que é um modo interativo, o *SparkSession* é criado automaticamente para o programador, e em uma aplicação Spark convencional, o programador deve criá-lo. Na Figura 21, o comando *spark.version* mostra a versão do Spark que está instalada. Na Figura 22, usamos o objeto *spark* para ler um arquivo-texto, mostrar as 10 primeiras linhas e contar o número de linhas.

Figura 22 – Lendo um arquivo texto por meio de um objeto *SparkSession*.



O *spark-shell* é um ambiente para programar na linguagem Scala; para usar o Python é necessário instalarmos o *PySpark*. Siga as instruções descritas no link: https://spark.apache.org/docs/latest/api/python/getting_started/install.html ou simplesmente execute *pip install pyspark* ou *pip3 install pyspark* (para Python 3).

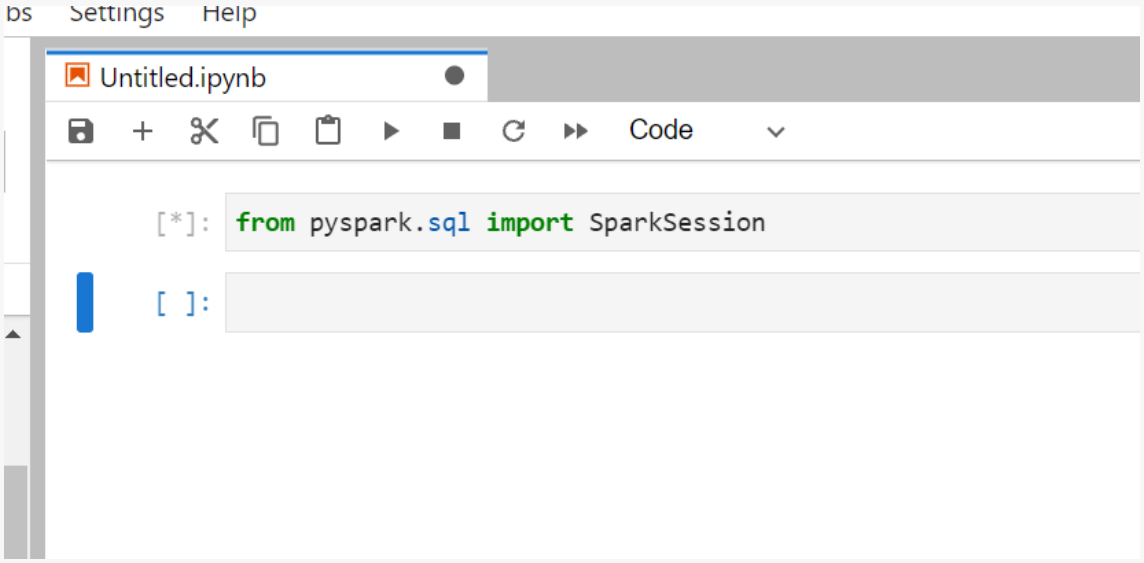
O *PySpark* é um ambiente interativo similar ao spark-shell, porém a linguagem suportada é o Python, como você pode ver na Figura 23.

Figura 23 – ambiente interativo do PySpark, útil para interagir com a API do Spark na linguagem Python.

```
pcalais:~/Downloads$ pyspark
Python 3.8.10 (default, Sep 28 2021, 16:10:42)
[GCC 9.3.0] on linux
Type "help", "copyright", "credits" or "license" for more information.
21/10/17 18:54:36 WARN Utils: Your hostname, pcalais-Inspiron-15-7000-Gaming resolves to a loopback
nterface wlp3s0)
21/10/17 18:54:36 WARN Utils: Set SPARK_LOCAL_IP if you need to bind to another address
21/10/17 18:54:37 WARN NativeCodeLoader: Unable to load native-hadoop library for your platform...
Using Spark's default log4j profile: org/apache/spark/log4j-defaults.properties
Setting default log level to "WARN".
To adjust logging level use sc.setLogLevel(newLevel). For SparkR, use setLogLevel(newLevel).
Welcome to
      ____              __
     / __/__  ___ _____/ /__
    _\ \/ _ \/ _ `/ __/  '_/
   /__ / .__/\_,_/_/ /_/\_\   version 3.1.2
      /_/

Using Python version 3.8.10 (default, Sep 28 2021 16:10:42)
Spark context Web UI available at http://192.168.0.11:4040
Spark context available as 'sc' (master = local[*], app id = local-1634507678142).
SparkSession available as 'spark'.
>>>
```

Por fim, é útil instalar um ambiente web para desenvolvimento interativo, como o JupyterLab (https://jupyter.org/install). Após instalá-lo, você poderá programar em Spark em Python em um ambiente interativo, conforme mostra a Figura 24.

Figura 24 – interface do Jupyter-Lab permite a criação de um Python Notebook.



Alternativa: usando uma imagem do Docker que já tem o Spark instalado.

Alguns alunos costumam ter dificuldade para instalar o Apache Spark no Linux ou no Windows. Ao invés de instalar o Spark na sua máquina,

você pode baixar uma imagem de uma instalação Linux que já contém o Spark instalado. O Docker é uma plataforma para criar, implantar e gerir aplicações em *containers*, que são componentes que contém aplicações junto com o sistema operacional e bibliotecas já instaladas. Para subir um container Docker que contém o sistema operacional Linux com o Spark já instalado, siga os passos:

1. Instale o Docker. No Windows, você pode seguir as instruções em: https://docs.docker.com/desktop/windows/install/.
2. Vá em https://hub.docker.com/r/bitnami/spark e siga as instruções:
    - Execute:
        - curl -L0 https://raw.githubusercontent.com/bitnami/bitnami-docker-spark/master/docker-compose.yml
    - Execute: docker-compose up
3. Agora você tem um container docker de pé que já tem o spark instalado.
4. Após subir o container via docker-compose up, encontre uma linha de log como esta: "spark-worker-2_1 | 21/11/11 00:30:05 INFO Worker: Successfully registered with master spark://156fbd78435c:7077"
5. A linha acima indica o ID do container. 156fbd78435c, no exemplo acima.
6. Entre no container: docker exec -ti *156fbd78435* /bin/bash.
7. Pronto, agora você já está dentro de uma máquina que tem o spark-shell e *pyspark* de pé e funcionando! Execute “pyspark” para testar.

8. Para copiar um arquivo da sua máquina para o container, você pode fazer: *docker cp titanic.csv 156fbd78435:/var/tmp/*, em que *156fbd78435* deve ser substituído pelo identificador que você encontrou no passo 4.

Contando palavras: O “Hello World” do Spark

Quando aprendemos uma nova linguagem de programação, quase sempre o primeiro programa que o livro ou o professor vai apresentar é o “Olá, Mundo!”. Aqui não vai ser diferente. Porém, o “Alô, Mundo!” do mundo de programação distribuída é um problema simples: dado um arquivo contendo um texto, devemos contar a frequência com que cada palavra ocorre neste documento. Note que o arquivo (ou conjunto de arquivos) pode ser potencialmente gigantesco – imagine petabytes – e portanto faria sentido lançar mão de uma plataforma de programação distribuída como o Spark.

Na instalação do Spark, a pasta *examples/src/main/python/* contém exemplos de aplicações em Spark na linguagem Python; *wordcount.py* é uma delas e está descrita na Figura 25. Execute você mesmo o programa pelo *pyspark* ou *jupyter-lab*, para que você adquira prática em criar e executar suas próprias aplicações.

Figura 25 – Aplicação Spark que conta palavras a partir de um arquivo texto.

```
1  import sys
2  from operator import add
3
4  from pyspark.sql import SparkSession
5
6
7  if __name__ == "__main__":
8      if len(sys.argv) != 2:
9          print("Usage: wordcount <file>", file=sys.stderr)
10         sys.exit(-1)
11
12     spark = SparkSession\
13         .builder\
14         .appName("PythonWordCount")\
15         .getOrCreate()
16
17     lines = spark.read.text(sys.argv[1]).rdd.map(lambda r: r[0])
18     counts = lines.flatMap(lambda x: x.split(' ')) \
19                   .map(lambda x: (x, 1)) \
20                   .reduceByKey(add)
21     output = counts.collect()
22     for (word, count) in output:
23         print("%s: %i" % (word, count))
24
25     spark.stop()
```

O programa pode ser descrito da seguinte forma:

- A Linha 12 constrói o objeto *SparkSession*;
- A Linha 17 lê o arquivo texto e armazena as linhas do arquivo no RDD *lines*;
- A linha 18 quebra cada linha em palavras e para cada palavra emite uma tupla que contém a palavra e contagem 1. Finalmente, uma transformação de redução invoca a função *add* para somar o valor das tuplas, cuja chave é a mesma. No caso, as tuplas são na forma (chave, valor), ou seja, (“spark”, 1) é uma tupla com chave “spark” e valor 1.

Por fim, a linha 21 traz a resposta da computação para o *driver*, a partir da ação *collect()*, e em seguida os resultados são impressos na saída padrão.

Bem-vindo ao Spark! Na próxima seção detalharemos as operações que foram usadas neste programa -- bem como as outras operações principais que compõem a API de RDDs do Apache Spark.

Operações: transformações e ações

RDDs são imutáveis: uma vez criados, nunca são alterados. A única forma de executar computações sobre RDDs é, portanto, *transformando* um RDD em outro, a partir de operações chamadas de *transformações*. As principais transformações fornecidas pela API de RDDs do Spark são:

- map(func);
- flatMap(func);
- filter(func);
- reduceByKey(func);
- sortByKey(func);
- union(rdd);
- intersection(rdd);
- distinct(rdd);
- join(rdd).

As *ações*, por outro lado, são operações que, a partir de um RDD, produz um valor que não é um RDD. Os valores produzidos por uma ação são copiados para o programa *driver* ou para o sistema de armazenamento de arquivos. As principais ações da API de RDDs do Spark são:

- foreach(func);
- collect();

- count();
- take(n);
- countByValue();
- reduce(func);
- saveAsTextFile(path).

Dataframes

O RDD é uma abstração simples e poderosa: uma coleção de objetos que é particionada em executores e pode executar computações sobre seus objetos de forma distribuída e paralela. Mas nem tudo é perfeito: como a função que o Spark executa em suas transformações é genérica e trata os objetos de forma opaca, o sistema não sabe, por exemplo, que o programador está acessando uma coluna específica de um objeto. Isso impede o Spark de executar otimizações de desempenho e torna o código que poderia ser mais simples em um código difícil e trabalhoso de ler.

A Figura 26 ilustra um código em Spark que lê tuplas <nome, idade> e objetiva computar, para cada nome, a idade média. Note, por exemplo, que *Pedro* ocorre duas vezes e a idade média deste nome é 39. O código não é muito legível: a transformação *map* é necessária duas vezes e é necessária uma redução para somar as idades.

Figura 26 – Computando a idade média por nome das pessoas usando a API de RDDs. A lógica dentro das transformações não é tão simples de entender.

```
In [16]:
# Agregate all ages by name and get the average name by age.

# Create an RDD of tuples (name, age)
dataRDD = spark.sparkContext.parallelize([("Pedro", 38), ("Maria", 20), ("Pedro", 40), ("Rafael", 10)])

agesRDD = (dataRDD
           .map(lambda x: (x[0], (x[1], 1)))
           .reduceByKey(lambda x, y: (x[0] + y[0], x[1] + y[1]))
           .map(lambda x: (x[0], x[1][0]/x[1][1])))

agesRDD.collect()

Out[16]: [('Pedro', 39.0), ('Maria', 20.0), ('Rafael', 10.0)]
```

O Spark, desde a versão 2.x, introduziu o *Dataframe*, uma API que fornece uma coleção distribuída de dados, mas, diferentemente do RDD lida com campos que podem ser colunas nomeadas. Além disso, o *Dataframe* permite que o programador opere sobre esses dados transformações típicas do mundo de análise de dados, como filtro, seleção, contagem, agregação e agrupamento dos dados por colunas.

Note, na Figura 27, como o código com a API de *Dataframes* é mais claro e possui uma maior expressividade, por meio de operadores de alto nível que dizem para o Spark diretamente o que ele precisa fazer. E temos um bônus aqui: como o Spark entende o que queremos fazer, ele pode otimizar e organizar as operações para otimizar a execução. O programador tem então duas opções: uma API expressiva que implementa operações recorrentes no mundo de análise de dados e a API de RDDs, de mais baixo nível e com menos estrutura, que dá mais liberdade.

Os DataFrames do Spark foram inspirados nos *Dataframes* do Pandas, uma biblioteca para análise de dados do Python, e funcionam como uma tabela distribuída e em memória colunas com nome e esquema. Cada coluna tem um tipo (inteiro, string, array, data, *timestamp* etc.) e um *Dataframe* funciona como uma tabela de um banco de dados.

Figura 27 – Computando a idade média por nome das pessoas usando a API *DataFrame*.

```
from pyspark.sql import SparkSession
from pyspark.sql.functions import avg

spark = (SparkSession.builder
         .appName("Ages")
         .getOrCreate())

# Create a DataFrame
data_df = spark.createDataFrame([("Pedro", 38), ("Maria", 20), ("Pedro", 40), ("Rafael", 10)], ["nome", "idade"])

avg_df = data_df.groupBy("nome").agg(avg("idade"))

avg_df.show()
```

```
+------+----------+
|  nome|avg(idade)|
+------+----------+
| Pedro|      39.0|
| Maria|      20.0|
|Rafael|      10.0|
+------+----------+
```

No Capítulo 4, vamos nos aprofundar no Spark SQL e nos aprofundarmos em mais detalhes nos usos e possibilidades dos *dataframes*.

> Capítulo 3

Capítulo 3. Estatística descritiva usando Apache Spark

Neste capítulo vamos rapidamente revisar o que é estatística descritiva e demonstrar como as principais métricas de descrição de dados podem ser calculados usando transformações e ações sobre RDDs

O que é estatística descritiva?

Estatística descritiva é um ramo da estatística que se dedica a sumarizar as características e atributos de um conjunto de dados. Genericamente, métricas de estatística descritiva pretendem obter, para uma população ou amostra:

- Medidas de tendência central, como:
  - Média.
  - Mediana.
  - Moda.
- Medidas de variabilidade, como:
  - Variância.
  - Desvio-padrão.
  - Intervalo (valor máximo e mínimo).
  - Percentis.
- Distribuição de frequência dos dados.

A estatística descritiva se contrapõe à estatística inferencial, que pretende generalizar, a partir de uma amostra, conclusões que valem para a população. A estatística inferencial lança mão de métodos como teste de

hipótese, análise de regressão e intervalos de confiança para construir modelos e realizar predições sobre o futuro.

A estatística descritiva é especialmente importante para cientistas de dados como forma de sumarizar e entender os dados com os quais vai se trabalhar, além de verificar e levantar hipóteses iniciais sobre os dados.

Na próxima seção, vamos aprender como podemos computar diversas medidas de estatística descritiva por meio de aplicações Spark.

Computando estatísticas descritivas com Spark

O Spark, por meio da API de Dataframes, já implementa as principais medidas de estatística descritiva. Vamos ver um exemplo. Primeiro, vamos usar o script da Figura 28 para gerar um milhão de registros com duas colunas aleatórias -- a primeira, seguindo uma distribuição uniforme e a segunda usando uma distribuição normal.

Figura 28 – Gerando números aleatórios.

```
from pyspark.sql.functions import rand, randn
from pyspark import SparkContext
from pyspark.sql import SQLContext

spark = SparkSession\
        .builder\
        .appName("PythonWordCount")\
        .getOrCreate()

sqlc = SQLContext(spark.sparkContext)

df = (sqlc.range(0, 1000 * 1000)
        .withColumn('uniform', rand(seed=10))
        .withColumn('normal', randn(seed=27)))

print('# rows: ', df.count())
df.show()

```

Podemos, agora, usar a função *describe()* para computar a média, desvio-padrão e valores mínimo e máximo das colunas, como ilustrado na Figura 29.

Figura 29 – Usando *describe()* para computar estatística descritiva com Spark.

```
df.describe().show()
```

```
+-------+------------------+--------------------+--------------------+
|summary|                id|             uniform|              normal|
+-------+------------------+--------------------+--------------------+
|  count|           1000000|             1000000|             1000000|
|   mean|          499999.5|  0.4997785318606761|6.545992003465573E-4|
| stddev|288675.27893234405|  0.2887560412263698|  1.0003498848232582|
|    min|                 0|2.710561290975022E-7|  -4.949492960499273|
|    max|            999999|  0.9999998822463074|   4.474351963425938|
+-------+------------------+--------------------+--------------------+
```

A correlação entre duas variáveis é uma medida que pode ser bastante útil quando estamos analisando dados de forma exploratória; no Spark é fácil computar esta métrica, conforme mostra a Figura 30.

Figura 30 – Computando a correlação estatística entre variáveis.

```
from pyspark.sql.functions import rand, randn
from pyspark import SparkContext
from pyspark.sql import SQLContext


spark = SparkSession\
        .builder\
        .appName("PythonWordCount")\
        .getOrCreate()

sqlContext = SQLContext(spark.sparkContext)

df = sqlContext.range(0, 1000 * 1000).withColumn('rand1', rand(seed=10)).withColumn('rand2', rand(seed=27))

print('cor(rand2, rand2): ', df.stat.corr('rand2', 'rand2'))

print('cor(rand1, rand2): ', df.stat.corr('rand1', 'rand2'))
```

```
cor(rand2, rand2):  1.0
cor(rand1, rand2):  0.00137206619523886
```

> Capítulo 4

Capítulo 4. Spark SQL

Nos próximos três capítulos vamos nos dedicar a conhecer os principais componentes do Spark que, quando plugados ao componente *core* do Spark (as APIs de RDDs e Dataframes), permitem criar aplicações de processamento de *big data* poderosas. Vamos começar com o Spark SQL.

SQL - *structured query language* - é uma linguagem criada na década de 1970 para ajudar programadores a gerir dados armazenados em bancos de dados relacionais, isto é, bancos de dados compostos por relações – uma relação é, grosso modo, uma tabela contendo linhas (registros) e colunas. Por exemplo, uma tabela "Funcionário" pode conter colunas como Nome, CPF e data de nascimento, e cada linha da tabela representa um funcionário.

A linguagem SQL permite ao programador manipular os dados armazenados em um banco de dados por meio de algumas operações:

- Adicionar registros;
- Remover registros;
- Atualizar registros;
- Ler registros (o Spark entra aqui!).

Por exemplo, uma consulta SQL simples que lê as colunas *Nome* e *CPF* de uma tabela Funcionário poderia ser escrita como:

SELECT Nome, CPF FROM Funcionario;

Onde entra o Spark nessa história? O Spark SQL é um módulo do Apache Spark que integra o processamento de dados estruturados e relacionais com a API do Spark, que, por sua vez, é altamente inspirada no paradigma de programação funcional. Ou seja, o Spark SQL traz para o Spark

a possibilidade dos programadores usarem o benefício de programação relacional (escrevendo consultas SQL) ao mesmo tempo em que podem usar bibliotecas complexas de aprendizado de máquina.

O Spark SQL permite:

- Conectar com fontes de dados externas como JDBC, PostgreSQL, MySQL, Tableau, Azure Cosmos DB e MS SQL Server.
- Trabalhar com tipos simples e complexos.
- O uso de funções definidas pelo usuário que interpretem, processem e transformem dados lidos do banco de dados.

Funções definidas pelo usuário

O Apache Spark possui uma lista grande de funções, porém ele é tão flexível que permite que Engenheiros e Cientistas de Dados criem suas próprias funções, chamadas de UDFs (User-Defined Functions). Essas funções podem ser utilizadas em conjunto com o Spark SQL. Nas Figuras 31 e 32, mostramos uma UDF que calcula o quadrado de um valor, e o Spark SQL permite que executemos uma consulta SQL que usa esta função.

Figura 31 – Funções definidas pelo usuário (UDFs) podem ser usadas com Spark SQL.

```
from pyspark.sql import SparkSession
from pyspark.sql.types import LongType

spark = SparkSession\
        .builder\
        .appName("SquareFunction")\
        .master("local[*]")\
        .getOrCreate()

# Create square function
def square(s):
     return s*s

# Register UDF
spark.udf.register("square", square, LongType())

# Generate temporary view
spark.range(1, 10000).createOrReplaceTempView("udf_test")

# Run query
spark.sql("SELECT id, square(id) from udf_test").show()
```

Figura 32 – Resultado da aplicação da função *square()* da Figura 18, usando Spark SQL.

```
+---+----------+
| id|square(id)|
+---+----------+
|  1|         1|
|  2|         4|
|  3|         9|
|  4|        16|
|  5|        25|
|  6|        36|
|  7|        49|
|  8|        64|
|  9|        81|
| 10|       100|
| 11|       121|
| 12|       144|
| 13|       169|
| 14|       196|
| 15|       225|
| 16|       256|
| 17|       289|
| 18|       324|
| 19|       361|
| 20|       400|
+---+----------+
only showing top 20 rows
```

Podemos observar aqui um dos poderes do Spark como plataforma unificada e flexível: um Cientista de Dados poderia criar um modelo de ML a partir de uma UDF e um Analista de Dados pode executar consultas com esta função sem ter que entender os detalhes do modelo.

Os objetivos do Spark SQL são:

- Suportar processamento relacional tanto em programas Spark quanto lendo de fontes externas.
- Prover desempenho usando técnicas estabelecidas de sistemas de banco de dados.
- Suportar novas fontes de dados com facilidade.
- Habilitar integração com aplicações de aprendizado de máquina e processamento de grafos.

O Spark SQL pode ler dados em vários formatos:

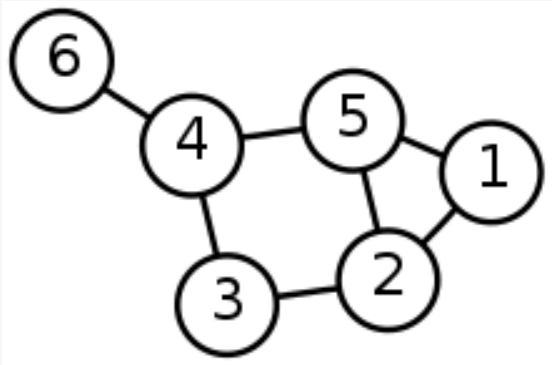
- CSV.
- JSON.
- Arquivos-texto.
- Parquet.
- Tabelas do Apache Hive, um armazém de dados que faz parte do ecossistema do Apache Hadoop.

> Capítulo 5

Capítulo 5. Spark GraphX

O grafo é uma das estruturas matemáticas mais fundamentais em Ciência da Computação. Um grafo é um conjunto de vértices e arestas, veja um exemplo na Figura 33:

Figura 33 – Um grafo é um conjunto de vértices e arestas.



Em muitas aplicações faz sentido representar não apenas os dados, mas as conexões entre eles: é o caso de redes sociais, sistemas de telecomunicação, páginas da web, artigos acadêmicos e suas citações. Nestes casos, modelar os dados como uma rede (grafo) permite resolver problemas de forma mais assertiva, como recomendação de conteúdo, detecção de fraude e ranqueamento de páginas da Web.

GraphX é o módulo do Spark que manipula grafos, como redes sociais, rotas, topologias de redes de computadores, redes de telecomunicação, redes biológicas etc.) e executa computações sobre os grafos em paralelo. Ele é especialmente útil quando o grafo não cabe na memória de uma única máquina, e oferece os principais algoritmos básicos que operam sobre grafos, como PageRank, contagem de triângulos e obtenção de componentes conectados.

> Capítulo 6

Capítulo 6. Spark MLLib

Até aqui muito da nossa dedicação foi sobre tarefas de engenharia de dados. Tipicamente, engenharia de dados é uma etapa anterior para preparar os dados para tarefas de aprendizado de máquina (ML). Atualmente, aplicações de aprendizado de máquina e inteligência artificial nos impactam diretamente, como recomendação online, propaganda, detecção de fraudes, reconhecimento de imagens, processamento de linguagem natural, classificação de texto e de spam, entre outras.

Em poucas palavras, aprendizado de máquina é a criação de modelos estatísticos, por meio de álgebra linear e otimização, que extrai padrões a partir dos dados e consegue realizar previsões sobre o futuro. Por exemplo, a partir de uma coleção de documentos de e-mail pré-classificados entre *spam* e não-*spam*, um modelo de aprendizado de máquina pode ser treinado para, ao ser apresentado a uma nova mensagem de e-mail, identificar se ela é um *spam* ou não. Outra aplicação típica de aprendizado de máquina é criar um classificador que, dado uma imagem, identifica se ela contém um gato ou um cachorro – ou, mais importante, se ela contém um bandido armado ou não, por exemplo.

O Spark MLLib é o módulo do Spark que fornece implementações dos principais algoritmos de aprendizado de máquina encontrados na literatura da área, em particular, algoritmos para:

- Classificação.
- Regressão.
- Agrupamento (*clustering*).

Nos problemas de **classificação**, os dados consistem em um conjunto de registros de entrada e cada um tem um rótulo associado a ele. O objetivo é prever qual o rótulo para um registro novo. Exemplos clássicos de problemas de classificação são:

- Uma foto contém a imagem de um gato ou cachorro?
- Este documento de texto está escrito em inglês, português ou espanhol?
- Dado a temperatura, direção do vento e umidade relativa do ar, vai chover hoje?

Os problemas de **regressão**, por sua vez, são similares, porém, o valor a ser previsto pelo modelo de aprendizado de máquina é um valor contínuo. Exemplos de problema de regressão incluem:

- Quantos sorvetes serão vendidos hoje, informada a temperatura e o dia da semana?
- Qual o preço de um apartamento, informado sua área, número de quartos, número de banheiros e bairro?

Finalmente, nos problemas de agrupamento (ou aprendizado não-supervisionado), não temos um valor a prever, mas queremos encontrar estruturas e regularidades nos dados, em particular dividi-lo em grupos. Exemplos clássicos são:

- Dadas as informações de idade, renda e sexo dos meus clientes, em quantas categorias posso dividi-los? Esta informação pode ser útil para guiar ações de marketing.
- Dado um conjunto de documentos de texto que versam sobre diversos temas, como podemos agrupar documentos que versem sobre o mesmo assunto?

O Spark oferece o pacote *spark.ml* para que Cientistas de Dados possam preparar os dados e construir modelos de aprendizado de máquina. Os algoritmos são implementados no Spark de modo que escalem linearmente de acordo com a quantidade de dados, permitindo criar modelos com grandes volumes de dados como entrada. Esta é uma vantagem sobre o *scikit-learn*, biblioteca de aprendizado de máquina do Python, que é projetada para executar em apenas um único nodo.

Vamos ver um exemplo de um código em Python que usa a SparkML. Você pode obter o notebook e o arquivo CSV que usaremos neste exemplo em: https://homepages.dcc.ufmg.br/~pcalais/IGTI/cientista_dados/spark/income-dataset/. Essa planilha contém informações gerais de trabalhadores, como sua idade, sexo e quantas horas por semana cada um deles trabalha. Uma das informações diz se a renda do trabalhador é inferior ou superior a 50 mil dólares (coluna *income*). Esse é um problema simples para praticarmos a Spark ML: será que conseguimos prever a renda dos trabalhadores a partir de informações como idade, sexo, cor da pele, entre outras? Sugiro que você escreva o código junto comigo para praticar e aprender fazendo.

O primeiro bloco de código você já conhece: criamos um objeto *SparkSession*, por meio do qual enviamos comandos para o Spark.

In [1]:
```
# Vamos criar o objeto sparkSession,
# que é nosso ponto de entrada de comunicação com a plataforma Spark.
from pyspark.sql import SparkSession

# Spark entry point
spark = SparkSession \
.builder \
.appName("Aula 6 - SparkML e Pipelines") \
.getOrCreate()

spark.version
```

Out[1]: '3.2.1'

No segundo bloco de código, lemos o arquivo CSV usando *spark.read.csv*, comando que você já utilizou no trabalho prático. Veja que o comando *printSchema()* nos mostra as colunas contidas na planilha CSV:

```
In [2]:
# Vamos ler o arquivo com os dados socioeconomicos
# Baixe em http://www.dcc.ufmg.br/~pcalais/IGTI/cientista_dados/spark/income-dataset
income_df = spark.read.csv('C:/Users/pedro.guerra/Downloads/income-dataset.csv',header='True',inferSchema='True')
income_df.printSchema()
```

```
root
 |-- age: integer (nullable = true)
 |-- workclass: string (nullable = true)
 |-- fnlwgt: integer (nullable = true)
 |-- education: string (nullable = true)
 |-- education_num: integer (nullable = true)
 |-- marital_status: string (nullable = true)
 |-- occupation: string (nullable = true)
 |-- relationship: string (nullable = true)
 |-- race: string (nullable = true)
 |-- sex: string (nullable = true)
 |-- capital_gain: integer (nullable = true)
 |-- capital_loss: integer (nullable = true)
 |-- hours_per_week: integer (nullable = true)
 |-- native_country: string (nullable = true)
 |-- income: string (nullable = true)
```

Use o comando *show()* para visualizar alguns dados. Por exemplo, *show(2)* vai te mostrar dois registros da planilha:

```
In [4]:
income_df.show(2)
```

```
+---+----------------+------+---------+-------------+------------------+---------------+-------------+-----+----+------------+-
----------+--------------+--------------+------+
|age|       workclass|fnlwgt|education|education_num|    marital_status|     occupation| relationship| race| sex|capital_gain|c
apital_loss|hours_per_week|native_country|income|
+---+----------------+------+---------+-------------+------------------+---------------+-------------+-----+----+------------+-
----------+--------------+--------------+------+
| 39|       State-gov| 77516|Bachelors|           13|     Never-married|   Adm-clerical|Not-in-family|White|Male|        2174|
0|            40| United-States| <=50K|
| 50|Self-emp-not-inc| 83311|Bachelors|           13|Married-civ-spouse|Exec-managerial|      Husband|White|Male|           0|
0|            13| United-States| <=50K|
+---+----------------+------+---------+-------------+------------------+---------------+-------------+-----+----+------------+-
----------+--------------+--------------+------+
only showing top 2 rows
```

Usando *describe()*, conforme você aprendeu na aula sobre estatística descritiva, você pode conhecer um pouco mais dos dados. Por exemplo, a idade média dos indivíduos deste conjunto de dados é 38,64, e o indivíduo mais velho tem 90 anos:

```
In [5]:
income_df.describe('age').show()
```

```
+-------+------------------+
|summary|               age|
+-------+------------------+
|  count|             48842|
|   mean| 38.64358543876172|
| stddev|13.710509934443603|
|    min|                17|
|    max|                90|
+-------+------------------+
```

Veja que, em média, o tempo médio de trabalho de cada pessoa é de cerca de 40 anos:

```
In [6]:  1 income_df.select('hours_per_week').summary().show()
```

```
+-------+------------------+
|summary|    hours_per_week|
+-------+------------------+
|  count|             48842|
|   mean|40.422382375824085|
| stddev|12.391444024252301|
|    min|                 1|
|    25%|                40|
|    50%|                40|
|    75%|                45|
|    max|                99|
+-------+------------------+
```

Será que conseguimos, a partir dos atributos dos trabalhadores, prever qual é sua renda?

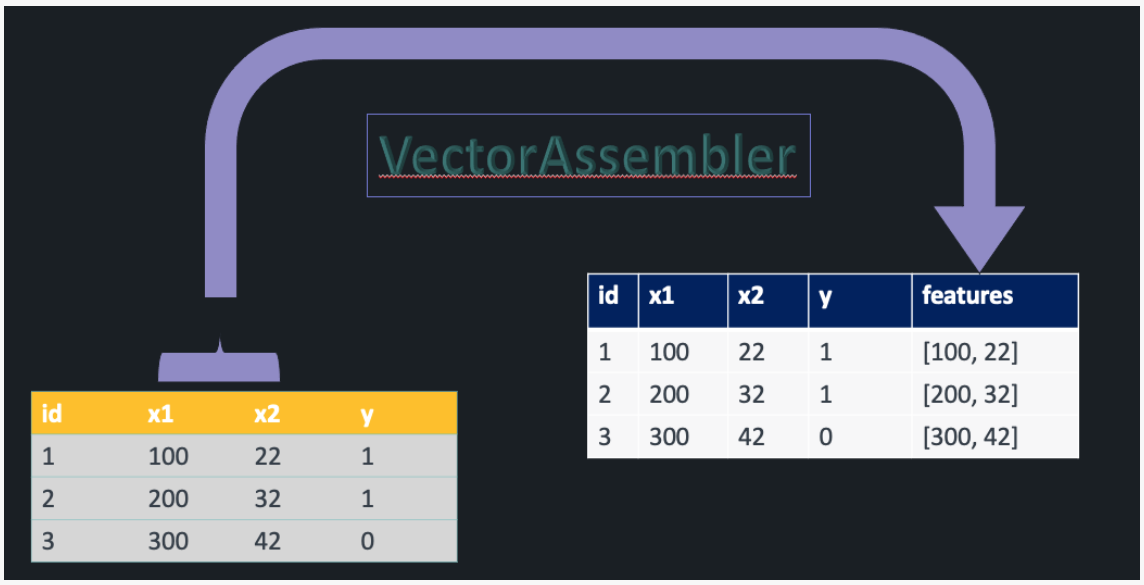
Vamos começar lidando com os atributos numéricos dos trabalhadores:

- Idade (*age*).
- Número de anos de estudo (*education_num).*
- Ganho de capital (*capital_gain*).
- Perda de capital (*capital_loss).*
- Número de horas de trabalho por semana (*hours_per_week).*

Note no bloco abaixo que, a partir dessas colunas, criamos um *VectorAssembler*. O papel do *VectorAssembler* é combinar uma lista de colunas em um único vetor coluna, pois é o formato que o algoritmo de aprendizado de máquina espera – ele não entende colunas de uma planilha CSV.

```
In [25]:
1  from pyspark.ml.feature import VectorAssembler
2
3  # Usa colunas numéricas para prever a renda.
4  numericCols = ["age", "education_num", "capital_gain", "capital_loss", "hours_per_week"]
5
6  # VectorAssembler é um transfomer
7  # Transforma um dataframe com colunas em um vetor com colunas
8  # [age | hours_per_week | education_num] => [20, 40, 13]
9  vecAssembler = VectorAssembler(inputCols=numericCols, outputCol="features")
```

Veja na figura abaixo a transformação que o *VectorAssembler* desempenha sobre os dados. Nenhuma mudança na natureza do dado acontece de verdade, estamos apenas adequando o dado ao formato que o código que aprende o modelo de aprendizado de máquina espera:



Vamos agora executar o algoritmo de aprendizado de máquina conhecido como Árvore de Decisão. Criamos um objeto do tipo *DecisionTreeClassifier*.

```
In [33]:
1  from pyspark.ml.classification import DecisionTreeClassifier
2
3  dtc = DecisionTreeClassifier(labelCol='income',featuresCol='features')
```

Os comandos abaixo criam um *Pipeline* (linha 4), dividem os dados em dois conjuntos (treino e teste, na linha 7), treina o modelo contra os dados de treino (linha 10) e, finalmente, executa as previsões de renda contra os dados de teste, na linha 13. Note que dividir os dados em dois

conjuntos (treino e teste) é importante, visto que, caso contrário, vamos avaliar o modelo de árvore de decisão contra os dados que o modelo observou para aprender os padrões de renda, e por isso esta avaliação não será justa.

```
In [32]:
1  from pyspark.ml import Pipeline
2
3  # Um pipelineé uma sequência de estágios
4  pipeline = Pipeline(stages=[vecAssembler, dtc])
5
6  # Separa os dados em dados de treinamento e teste
7  train_data,test_data = income_df.randomSplit([0.7,0.3])
8
9  # Pipeline é estimador que recebe um dataframe e produz um Model.
10 pipelineModel = pipeline.fit(train_data)
11
12 # Aplica o modelo do pipeline aos dados de teste.
13 predictionsDF = pipelineModel.transform(test_data)
```

Execute o código e você verá um erro como este:

```
    135             else:
click to scroll output; double click to hide
~\AppData\Local\Programs\Python\Python310\lib\site-packages\pyspark\sql\utils.py in raise_from(e)

IllegalArgumentException: requirement failed: Column income must be of type numeric but was actually of type string.
```

O que aconteceu aqui? Novamente, temos um problema prático de formato de dados. Algoritmos de aprendizado de máquina não funcionam bem com colunas que contém texto.

Veja os valores distintos da coluna de renda – *income*:

```
In [34]:
1  income_df.select('income').distinct().show()
+------+
|income|
+------+
| <=50K|
|  >50K|
+------+
```

O *pyspark* se queixou que ela tem valores *String,* que não são numéricos. Vamos transformar a coluna em uma coluna numérica *income_label* usando *StringIndexer.* Essa função vai assinalar um número para cada valor distinto de *income*. Por exemplo: “<=50K” pode ser transformado no valor 0 e “>50K” pode ser transformado no valor 1. Nada muda na prática, mas estamos preparando os dados para um formato que o algoritmo de aprendizado de máquina entende.

In [36]:
```
from pyspark.ml.feature import StringIndexer

# A categoria que queremos prever tem dois valores que sÃ£o strings, '<=50K, and >50K'
# Usamos o estimador 'StringIndexer' para convertê-lo em um valor numérico.
labelToIndex = StringIndexer(inputCol="income", outputCol="income_label")

```

Vamos, agora, executar o pipeline novamente, com um novo estágio, *labelToIndex*.

In [37]:
```
from pyspark.ml import Pipeline

# Um pipeline é uma sequência de estágios
dtc = DecisionTreeClassifier(labelCol='income_label',featuresCol='features')
pipeline = Pipeline(stages=[labelToIndex, vecAssembler, dtc])

# Separa os dados em dados de treinamento e teste
train_data,test_data = income_df.randomSplit([0.7,0.3])

# Pipeline é estimador que recebe um dataframe e produz um Model.
pipelineModel = pipeline.fit(train_data)

# Aplica o modelo do pipeline aos dados de teste.
predictionsDF = pipelineModel.transform(test_data)
```

Sem erros, agora! Vamos computar a qualidade do modelo. O valor que você vai encontrar vai diferir um pouco, por causa da aleatoriedade na separação dos dados entre os conjuntos de treino e teste.

```
from pyspark.ml.evaluation import MulticlassClassificationEvaluator

# acurÃ¡cia: % de previsÃµes corretas
evaluator = MulticlassClassificationEvaluator(metricName="accuracy", labelCol='income_label')
print(f"Acurácia: {evaluator.evaluate(predictionsDF)}")
```

```
Acurácia: 0.8161640693516832
```

Vamos, agora, tentar melhorar a qualidade do modelo usando também os atributos categóricos, como educação, ocupação, raça, sexo,

entre outros. Veja no bloco de código abaixo que aplicamos *StringIndexer* sobre cada um destes campos (para transformar cada valor de cada coluna em um número sequencial) e, em seguida, aplicar *OneHotEncoder*, uma outra transformação que cria uma coluna para cada um destes valores. Por exemplo, imagine que *sexo* tem três valores *String*, *homem, mulher e não-informado*. O *StringIndexer* vai converter estes dois valores para números, como 0 para homem, 1 para mulher e 2 para *não-informado*. O algoritmo poderia trabalhar com esses valores, porém ele erroneamente consideraria que há uma informação de ordem nos valores, o que não é verdade: os valores *não-informado* e *mulher* não são maiores nem menores que *homem*; não existe esta noção de ordem. Para evitar confundir o algoritmo, usamos *one-hot encoding*, o que significa criar uma coluna "Sexo-Homem" que é 1 apenas para os registros dos homens, uma coluna "Mulher" que é 1 apenas para os registros das mulheres, e assim para todos os valores. Isso representa de forma mais fiel o dado para que o algoritmo de aprendizado de máquina tente extrair padrões. Veja o código. ("OHE" significa *One-Hot-Encoding)*.

```
from pyspark.ml.feature import StringIndexer, OneHotEncoder

categoricalCols = ["workclass", "education", "marital_status", "occupation", "relationship", "race", "sex"]

# Cria estimadores (que implementam fit()) que retornam funcoes que vao ser aplicadas para transformar o dataset.
stringIndexer = StringIndexer(inputCols=categoricalCols, outputCols=[x + "Index" for x in categoricalCols])
oneHotEncoder = OneHotEncoder(inputCols=stringIndexer.getOutputCols(), outputCols=[x + "OHE" for x in categoricalCols])

allCols = [c + "OHE" for c in categoricalCols] + numericCols
vecAssembler = VectorAssembler(inputCols=allCols, outputCol="features")
```

Ao construirmos e avaliarmos o novo modelo, veja que a qualidade das previsões melhorou um pouco, após adicionarmos os atributos categóricos:

In [47]:
```
from pyspark.ml.classification import DecisionTreeClassifier
dtc = DecisionTreeClassifier(labelCol='income_label',featuresCol='features')
```

In [50]:
```
from pyspark.ml import Pipeline

# Um pipeline Ã© uma sequÃªncia de estÃ¡gios
# Ã‰ um estimador
pipeline = Pipeline(stages=[stringIndexer, oneHotEncoder, labelToIndex, vecAssembler, dtc])

# Separa os dados em dados de treinamento e teste
train_data,test_data = income_df.randomSplit([0.7,0.3])

# Pipeline Ã© um estimador que recebe um dataframe e produz um Model.
pipelineModel = pipeline.fit(train_data)

# Aplica o modelo do pipeline aos dados de teste.
predictionsDF = pipelineModel.transform(test_data)
```

In [51]:
```
from pyspark.ml.evaluation import MulticlassClassificationEvaluator

# acurácia: % de previsões corretas
evaluator = MulticlassClassificationEvaluator(metricName="accuracy", labelCol='income_label'
print(f"Acurácia: {evaluator.evaluate(predictionsDF)}")
```

```
Acurácia: 0.8456021650879567
```

Pronto! Você usou a Spark ML para construir e avaliar seu primeiro modelo de aprendizado de máquina por meio do Apache Spark. Seu modelo é capaz de acertar, com 84% de acurácia, se a renda de um trabalhador é baixa ou alta.

> Capítulo 7

Capítulo 7. Spark Streaming

Dados, tradicionalmente, são processados em dois formatos: *batch* e *stream*. Vamos diferenciá-los.

Dados em batch são coletados ao longo de um intervalo de tempo; depois de coletados, são enviados para processamento, que pode ser demorado. Dados em stream, por outro lado, são produzidos continuamente e são processados um a um, ou em partes; tipicamente, seu processamento deve ser rápido pois o resultado é necessário imediatamente.

Exemplos típicos de dados que são produzidos como *streams,* incluem:

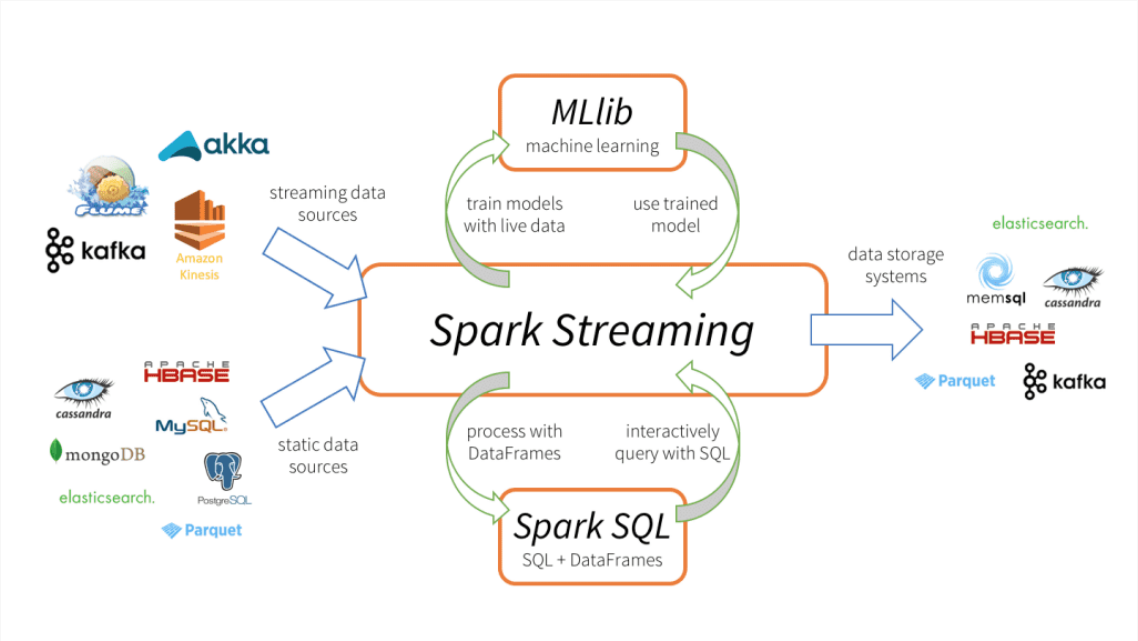
- Dados financeiros, por exemplo, a variação no preço das ações é um fluxo contínuo de dados. Modelos podem ser construídos para interpretar esses dados em tempo real e tomar decisões de investimento.
- Dados de logs de sistemas, que podem ser monitorados em tempo real para descoberta precoce de tentativas de fraude e ataques, como a negação de serviço.
- Dados de sites web produzem um *clickstream*, isto é, a sequência de cliques dos usuários, que é inerentemente um fluxo infinito.

O *Spark Streaming* é uma extensão da API básica do Spark que permite que Cientistas e Engenheiros de Dados processem dados em tempo real de múltiplas fontes de dados, e então possam ser direcionados para dashboards, bancos de dados e sistemas de arquivos.

O *Spark Streaming* pretende cumprir quatro objetivos principais:

- Ser capaz de se recuperar rapidamente de falhas e lentidões.
- Otimizar o uso de recursos por meio de balanceamento de carga.
- Permitir ao programador combinar dados em *stream* e *batch*.
- Se integrar com as demais APIs do Spark: Spark ML, Spark GraphX e Spark SQL.

Figura 34 – Spark Streaming permite a construção de aplicações complexas.



Fonte: databricks.com

> Capítulo 8

Capítulo 8. Deploy de uma aplicação Spark em um ambiente de cluster

Nos capítulos anteriores, aprendemos os conceitos-chave do Spark, bem como seus principais módulos, componentes e casos de uso. Nos nossos exemplos, o Spark era sempre executado no modo *local*, isto é, tanto o programa *driver*, os executores e o gestor do cluster executavam na mesma JVM e mesmo *host*.

Quando realmente queremos distribuir a computação em múltiplos nodos e nos beneficiarmos de toda a capacidade de escalabilidade do Spark, precisaremos executar a aplicação no modo *cluster*. Neste modo, usaremos um *gestor de cluster*, que pode ser:

- *standalone*: um gestor de cluster simples do próprio Spark.
- *Hadoop YARN (Yet Another Resource Negotiator)*: o gestor de recursos da plataforma Hadoop.
- Kubernetes: um sistema de orquestração de *containers* que automatiza o deploy e gestão de aplicações que executam em *containers* como o Docker.

Um passo importante para criarmos aplicações Spark que realmente executem em um cluster é o processo de implantação (deploy) do Spark em um sistema de cluster. O pyspark e o spark-shell, que usamos no decorrer do nosso estudo sobre as funcionalidades do Spark, são utilitários do tipo REPL (Read-Eval-Print-Loop) para que o programador execute e avalie seu código enquanto ele programa. Para executar uma aplicação Spark em um modo não-interativo, usamos o *spark-submit*, cujos detalhes sobre sua parametrização podem ser encontrados em: https://spark.apache.org/docs/latest/submitting-applications.html.

Figura 35 – O script *spark-submit* executa uma aplicação Spark em um ambiente de cluster.

```
./bin/spark-submit \
  --class <main-class> \
  --master <master-url> \
  --deploy-mode <deploy-mode> \
  --conf <key>=<value> \
  ... # other options
  <application-jar> \
  [application-arguments]
```

Referências

APACHE SOFTWARE FOUNDATION. **Spark**. Wilmington, DE, 2022. Disponível em: https://spark.apache.org/. Acesso em: 01 fev. 2022.

ARMBRUST, Michael *et al.* Spark SQL: relational data processing in Spark. *In:* ACM SIGMOD International Conference on Management of Data, 2015, Melbourne. **SIGMOD'15:** Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data. New York: Association for Computing Machinery, 2015. Disponível em: https://dl.acm.org/doi/pdf/10.1145/2723372.2742797. Acesso em: 31 jan. 2022.

DAMJI, Jules. S. **Learning Spark:** Lightning-Fast Data Analytics. Sebastopol, CA: O'Reilly Media, 2020.

DEAN, Jeffrey; GHEMAWAT, Sanjay. MapReduce: simplified data processing on large clusters. Communications of the ACM, New York, v. 51, n. 1, p. 107-113, jan. 2018. Disponível em: https://static.googleusercontent.com/media/research.google.com/pt-BR//archive/mapreduce-osdi04.pdf. Acesso em: 31 jan. 2022.

ZAHARIA, Matei *et al.* Resilient Distributed datasets: a fault-tolerant abstraction for in-memory cluster Computing. *In*: USENIX CONFERENCE ON NETWORKED SYSTEMS DESIGN AND IMPLEMENTATION, 9, 2012, San Jose, CA. **NSDI'12:** Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation. Berkeley, CA: USENIX Association, 2012. Disponível em: https://www.usenix.org/system/files/conference/nsdi12/nsdi12-final138.pdf. Acesso em: 31 jan. 2022.

ZAHARIA, Matei *et al.* Apache Spark: a unified engine for big data processing. **Communications of the ACM**, New York, v.59, n. 11, p. 56-65, nov. 2016. Disponível

em: https://cs.stanford.edu/~matei/papers/2016/cacm_apache_spark.pdf. Acesso em: 01 fev. 2022.

ZAHARIA, Matei *et al.* Spark: cluster Computing with working sets. *In:* USENIX CONFERENCE ON HOT TOPICS IN CLOUD COMPUTING, 2, 2010, Boston. HotCloud'10: Proceedings of the USENIX Conference on Hot topics in Cloud Computing. Berkeley, CA: USENIX Association, 2010. Disponível em: https://www.usenix.org/legacy/event/hotcloud10/tech/full_papers/Zaharia.pdf. Acesso em: 31 jan. 2022.