Semana 8

Objetivo: Introdução ao Apache Hadoop e Apache Spark.

Conteúdo:

- Apache Hadoop
- Apache Spark
- Spark vs Hadoop

Desafio: Realizar a leitura do material completo e executar os 2 exercícios.

Apache Hadoop

Antes de falarmos sobre o Spark precisamos falar sobre o Hadoop para conseguirmos fazer a distinção correta entre os frameworks.

O Apache Hadoop ou apenas Hadoop é um framework Open Source, assim como o Spark, que permite gerenciar e processar big data com eficiência em um ambiente de computação distribuído. Vamos começar tendo o entendimento do histórico do Hadoop até 2012 de acordo com o blog da Devmedia[4]:

- Fevereiro de 2003: Jeffrey Dean e Sanjay Ghemawat, dois engenheiros do Google, desenvolvem a tecnologia **MapReduce**, que possibilitou otimizar a indexação e catalogação dos dados sobre as páginas Web e suas ligações. O **MapReduce** permite dividir um grande problema em vários pedaços e distribuí-los em diversos computadores. Essa técnica deixou o sistema de busca do Google mais rápido mesmo sendo executado em computadores convencionais e menos confiáveis, diminuindo assim os custos ligados à infraestrutura;
- Outubro de 2003: O Google desenvolve o **Google File System**, um sistema de arquivos distribuído o **GoogleFS** (depois chamado de GFS), criado para dar suporte ao armazenamento e processamento do grande volume de dados da tecnologia MapReduce;
- Dezembro de 2004: o Google publica o artigo **Simplified Data Processing on Large Clusters**, de autoria dos engenheiros Dean e Ghemawat, onde eles apresentam os principais conceitos e características da tecnologia MapReduce, porém, sem detalhes sobre a implementação;
- Dezembro de 2005: o consultor de software Douglas Cutting divulgou a implementação de uma versão do MapReduce e do sistema de arquivos distribuídos com base nos artigos do **GFS** e do **MapReduce** publicados pelos engenheiros do Google. A implementação faz parte do subprojeto **Nutch**, adotado pela comunidade de software livre para criar um motor de busca na Web, normalmente denominado web crawler (um software que automatiza a indexação de páginas) e um analisador de formato de documentos parser . Tempos depois o Nutch seria hospedado como o projeto **Lucene**, na Apache Software Foundation , tendo como principal função fornecer um poderoso mecanismo de busca e indexação de documentos armazenados em diversos formatos, como arquivos de texto, páginas web, planilhas eletrônicas, ou qualquer outro formato do qual se possa extrair informação textual;
- Fevereiro de 2006: a empresa Yahoo! decide contratar Cutting e investir no projeto Nutch, mantendo o código aberto. Nesse mesmo ano, o projeto recebe o nome de **Hadoop**, passando a ser um projeto independente da Apache Software Foundation;
- Abril de 2007: o Yahoo! anuncia ter executado com sucesso uma aplicação Hadoop em um cluster de 1.000 máquinas. Também nessa data, o Yahoo! passa a ser o maior patrocinador do projeto. Alguns anos depois, a empresa já contava com mais de **40.000 máquinas executando o Hadoop** (White, 2010);
- Janeiro de 2008: o Apache Hadoop, na versão 0.15.2, amadurece como um projeto incubado na fundação Apache, e torna-se um dos principais projetos abertos da organização;

- Julho de 2008: uma aplicação Hadoop em um dos clusters do Yahoo! quebra o **recorde mundial de velocidade de processamento** na ordenação de 1 terabyte de dados. O cluster era composto de 910 máquinas e executou a ordenação em 209 segundos, superando o recorde anterior que era de 297 segundos;
- Setembro de 2009: a empresa Cloudera, especializa em Bigdata, contrata Cutting como líder do projeto. Cloudera é uma empresa que redistribui uma versão comercial derivada do Apache Hadoop;
- Dezembro de 2011: passados seis anos desde seu lançamento, o **Apache Hadoop** disponibiliza sua versão estável (a **1.0.0**). Entre as melhorias, destaca-se o uso do protocolo de autenticação de rede Kerberos, para maior segurança de rede; a incorporação do subprojeto HBase, oferecendo suporte a BigTable; e o suporte à interface WebHDFS, que permite o acesso HTTP para leitura e escrita de dados;
- Maio de 2012: a Apache faz o lançamento da versão da **2.0 do Hadoop**, incluindo alta disponibilidade no sistema de arquivos (**HDFS**) e melhorias no código.

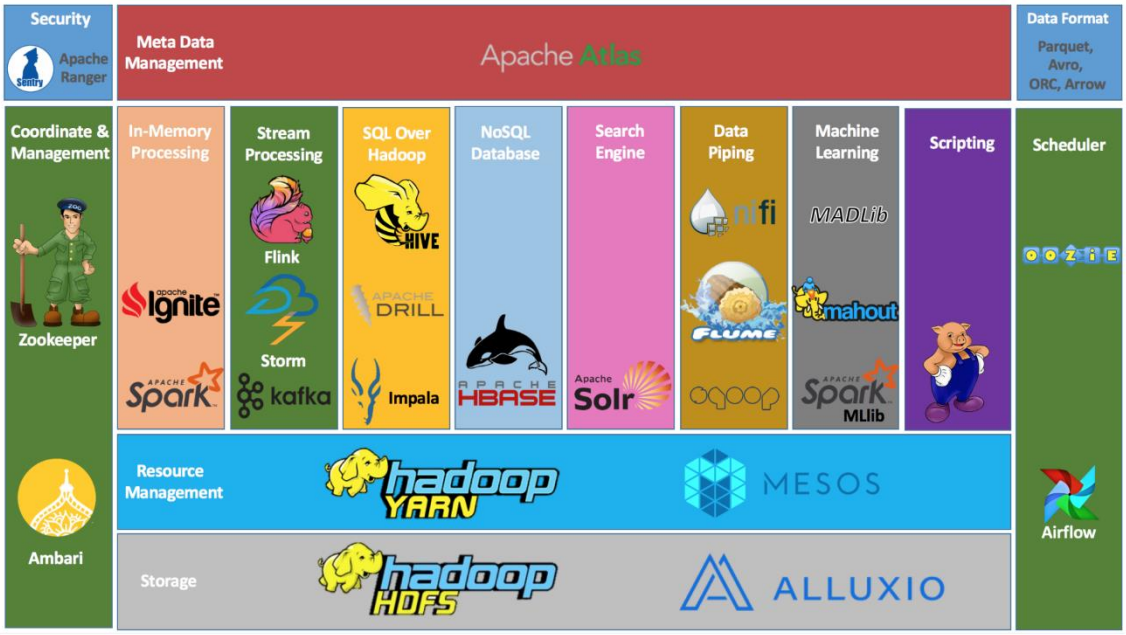
O Hadoop consiste em 4 (quatro) módulos principais:

1. **Hadoop Distributed File System (HDFS):** Os dados residem no sistema de arquivos distribuído do Hadoop, que é semelhante ao de um sistema de arquivos local em um computador típico. O HDFS é um sistema de armazenamento que usa tecnologia de armazenamento de objetos (object storage) armazenando os dados como objetos de tamanho variável, ao contrário do armazenamento tradicional de arquivos em blocos. Assim, são características do *object storage*: a durabilidade, a alta disponibilidade, a replicação, e a elasticidade, permitindo que a capacidade de armazenamento seja virtualmente infinita. No *object storage* cada item armazenado é um objeto definido por um identificador único, oferecendo uma alternativa ao modelo de arquivos baseados em blocos de dados. Por causa dessas facilidades, o armazenamento de dados na nuvem pode ser feito por meio do *object storage*. Seguindo essa tendência, os principais provedores de nuvem têm suas implementações de object storage, como o AWS S3, o Oracle Object Storage, o Azure Blob Storage e o Google Cloud Storage[5]. O HDFS oferece melhor rendimento de dados quando comparado aos sistemas de arquivos tradicionais. Além disso, o HDFS oferece excelente escalabilidade. Você pode dimensionar de uma única máquina a milhares com facilidade e em hardware comum.
2. **Yet Another Resource Negotiator (YARN):** facilita tarefas agendadas, gerenciamento completo e monitoramento de nós de cluster entre outros recursos.
3. **MapReduce**: O módulo Hadoop MapReduce ajuda os programas a realizar computação paralela de dados. A tarefa Map de MapReduce converte os dados de entrada em pares chave-valor. Reduzir tarefas consome a entrada, agrega-a e produz o resultado. Abstrai toda a computação paralela em apenas duas funções: Map e Reduce
4. **Hadoop Common:** O Hadoop Common usa bibliotecas Java que são padrões em todos os outros módulos.

Os componentes chave do Hadoop são o modelo de programação MapReduce e o sistema de arquivos distribuído HDFS. Entretanto, em meio a sua evolução, novos subprojetos, que são incorporados como componentes à arquitetura Hadoop, completam a infraestrutura do framework para resolver problemas específicos. Na camada de armazenamento de dados há o sistema de arquivos distribuído Hadoop Distributed File System (HDFS), um dos principais componentes do framework. Já na camada de processamento de dados temos o MapReduce,

que também figura como um dos principais subprojetos do Hadoop. Na camada de acesso aos dados são disponibilizadas ferramentas como Pig, Hive, Avro, Mahout, entre outras. Estas ferramentas tendem a facilitar a análise e consulta dos dados, fornecendo uma linguagem de consulta similar às utilizadas em bancos de dados relacionais (como a SQL, por exemplo). Assim, todo um ecossistema em volta do Hadoop é criado com ferramentas que suprem necessidades específicas; por exemplo, ZooKeeper, Flume e Chukwa, que melhoram a camada de gerenciamento. Essas ferramentas fornecem uma interface com o usuário que busca diminuir as dificuldades encontradas no manuseio das aplicações que rodam nessa plataforma[4]. Estas ferramentas complementares do Hadoop juntamente com os 4 principais módulos do Hadoop utiliza-se a denominação de Ecossistema. Abaixo uma imagem que contempla a descrição acima do Ecossistema Hadoop:

Ou, por exemplo, essa outra representação mais completa com outras soluções complementares:



Arquitetura dos componentes básicos do Hadoop

Toda a arquitetura do Hadoop tem objetivo de que aplicações MapReduce funcionem de forma distribuída em um cluster de máquinas, organizadas em uma máquina mestre e várias escravo. Aqui vamos focar em como componentes básicos, estamos falando do HDFS e do MapReduce, funcionam nessa arquitetura.

Para que o Hadoop funcione, é necessários cinco processos: `NameNode`, `DataNode`, `SecondaryNameNode`, `JobTracker` e `TaskTracker`. Os três primeiros são integrantes do modelo de programação **MapReduce**, e os dois últimos do sistema de arquivo **HDFS**. Os componentes NameNode, JobTracker e SecondaryNameNode são únicos para toda a aplicação, enquanto que o DataNode e JobTracker são instanciados para cada máquina do cluster[4].

HDFS (Hadoop Distributed File System)

Como dito anteriormente o HDFS é um sistema de arquivos distribuído é responsável pela organização, armazenamento, localização, compartilhamento e proteção de arquivos que estão distribuídos em computadores de uma rede. De acordo com a Devmedia[4], em sistemas distribuídos, quando mais de um usuário tenta gravar um mesmo arquivo simultaneamente, é necessário um controle da concorrência (acesso simultâneo ao mesmo recurso) para que haja uma operação atômica dos processos a fim de garantir a consistência das informações. Neste caso, um sistema de arquivos distribuídos deve garantir a atomicidade nas operações de leitura, escrita, criação ou remoção de um arquivo, de forma transparente para quem manipula os dados, como se fosse similar a um sistema de arquivos local.

Nota: Um sistema de arquivos é um componente do sistema operacional que permite ao usuário interagir com os arquivos e diretórios, seja para salvar, modificar ou excluir arquivos e diretórios (pastas), bem como instalar, executar ou configurar programas. Um sistema de arquivos distribuído faz tudo isso, mas em um ambiente de rede, onde os arquivos estão

fisicamente espalhados em máquinas distintas. Para quem usa tais arquivos, o sistema deve permitir as mesmas facilidades de um sistema de arquivos local.

O HDFS atua como um sistema de arquivos distribuído, localizado na camada de armazenamento do Hadoop, sendo otimizado para alto desempenho na leitura e escrita de grandes arquivos (acima dos gigabytes) que estão localizados em computadores (nós) de um `cluster`.

Dentre as características do HDFS estão a escalabilidade e disponibilidade graças à replicação de dados e tolerância a falhas. O sistema se encarrega de quebrar os arquivos em partes menores, normalmente blocos de 64MB, e replicar os blocos um número configurado de vezes (pelo menos três cópias no modo `cluster`, e um no modo `local`) em servidores diferentes, o que torna o processo tolerante a falhas, tanto em hardware quanto em software.

O fato é que cada servidor tem muitos elementos com uma probabilidade de falha alta, o que significa que sempre haverá algum componente do HDFS falhando. Por serem críticas, falhas devem ser detectadas de forma rápida e eficientemente resolvidas a tempo de evitar paradas no sistema de arquivos do Hadoop.

A arquitetura do HDFS é estruturada em `master-slave` (mestre-escravo), com dois processos principais, que são:

- **Namenode**: responsável por gerenciar os dados (arquivos) armazenados no HDFS, registrando as informações sobre quais datanodes são responsáveis por quais blocos de dados de cada arquivo, organizando todas essas informações em uma tabela de metadados. Suas funções incluem mapear a localização, realizar a divisão dos arquivos em blocos, encaminhar os blocos aos nós escravos, obter os metadados dos arquivos e controlar a localização de suas réplicas. Como o NameNode é constantemente acessado, por questões de desempenho, ele mantém todas as suas informações em memória. Ele integra o sistema HDFS e fica localizado no nó mestre da aplicação, juntamente com o JobTracker;
- **Datanode**: responsável pelo armazenamento do conteúdo dos arquivos nos computadores escravos. Como o HDFS é um sistema de arquivos distribuído, é comum a existência de diversas instâncias de DataNode em uma aplicação Hadoop, permitindo que os arquivos sejam particionados em blocos e então replicados em máquinas diferentes. Um DataNode poderá armazenar múltiplos blocos, inclusive de diferentes arquivos, entretanto, eles precisam se reportar constantemente ao NameNode, informando-o sobre as operações que estão sendo realizadas nos blocos.

MapReduce

O MapReduce é um modelo computacional para processamento paralelo das aplicações. De forma simples e resumida, ele abstrai as dificuldades do trabalho com dados distribuídos, eliminando quaisquer problemas que o compartilhamento de informações pode trazer em um sistema dessa natureza. Pode ser resumido nas seguintes fases:

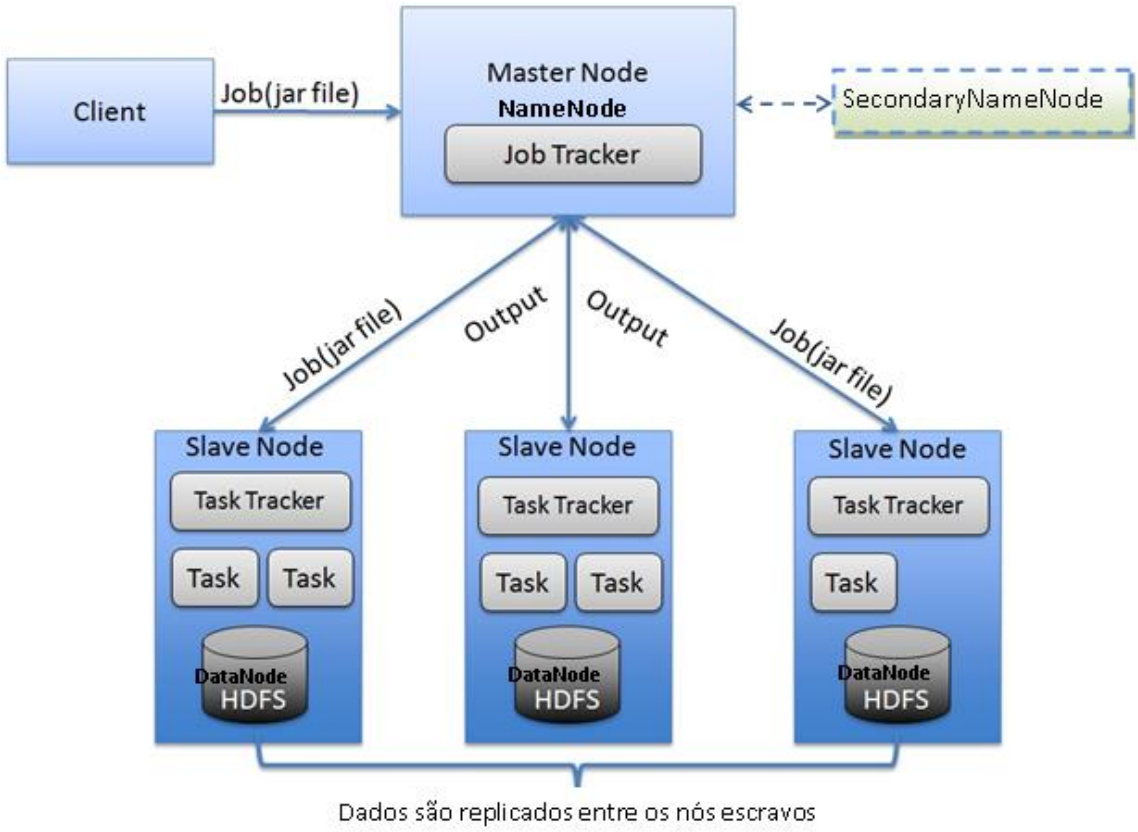
- **Map**: Responsável por receber os dados de entrada, estruturados em uma coleção de pares chave/valor. Tal função **map** deve ser codificada pelo desenvolvedor, através de programas escritos em Java ou em linguagens suportadas pelo Hadoop;

- **Shuffle**: A etapa de **shuffle** é responsável por organizar o retorno da função Map, atribuindo para a entrada de cada **Reduce** todos os valores associados a uma mesma chave;
- **Reduce**: Por fim, ao receber os dados de entrada, a função **Reduce** retorna uma lista de chave/valor contendo zero ou mais registros, semelhante ao **Map**, que também deve ser codificada pelo desenvolvedor.

A arquitetura do MapReduce segue o mesmo princípio master-slave, necessitando de três processos que darão suporte à execução das funções map e reduce do usuário, a saber:

- **JobTracker**: recebe a aplicação MapReduce e programa as tarefas map e reduce para execução, coordenando as atividades nos TaskTrackers. Sua função então é designar diferentes nós para processar as tarefas de uma aplicação e monitorá-las enquanto estiverem em execução. Um dos objetivos do monitoramento é, em caso de falha, identificar e reiniciar uma tarefa no mesmo nó, ou, em caso de necessidade, em um nó diferente;
- **TaskTracker**: processo responsável por executar as tarefas de map e reduce e informar o progresso das atividades. Assim como os DataNodes, uma aplicação Hadoop é composta por diversas instâncias de TaskTrackers, cada uma em um nó escravo. Um TaskTracker executa uma tarefa map ou uma tarefa reduce designada a ele. Como os TaskTrackers rodam sobre máquinas virtuais, é possível criar várias máquinas virtuais em uma mesma máquina física, de forma a explorar melhor os recursos computacionais;
- **SecondaryNameNode**: utilizado para auxiliar o NameNode a manter seu serviço, e ser uma alternativa de recuperação no caso de uma falha do NameNode. Sua única função é realizar pontos de checagem (checkpointing) do NameNode em intervalos pré-definidos, de modo a garantir a sua recuperação e atenuar o seu tempo de reinicialização.

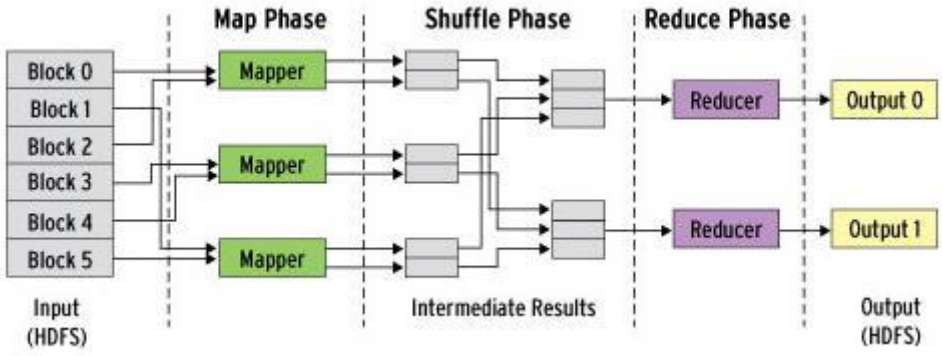
Abaixo pode-se observar como os processos da arquitetura do Hadoop estão interligados, organizados em nós mestre e escravos. O mestre contém o NameNode, o JobTracker e possivelmente o SecondaryNameNode. Já a segunda camada, constituída de nós escravos, comporta em cada uma de suas instâncias um TaskTracker e um DataNode, vinculados respectivamente ao JobTracker e ao NameNode do nó mestre. Uma tarefa (task) que roda em um nó escravo pode ser tanto de uma função map quanto de uma função reduce.



Algoritmo de MapReduce

De acordo com a publicação da Admin Network & Security – Discover Hadoop[6], o principal pilar de um sistema de MapReduce é um sistema de arquivos distribuído cuja funcionalidade básica é explicada fácil: arquivos grandes são divididos em blocos de tamanho igual, que são distribuídos pelo cluster para armazenamento. Porque você sempre precisa considerar a falha do computador em um cluster maior, cada bloco é armazenado várias vezes (tipicamente três vezes) em computadores diferentes. Aqui podemos perceber que eles estão falando do HDFS.

Na implementação do MapReduce, o usuário aplica uma sucessão alternada de funções **map** e **reduce** aos dados. Execução paralela dessas funções, e as dificuldades que ocorrem no processo, são tratadas automaticamente pelo framework. **Uma** iteração compreende três fases: map, shuffle e reduce, conforme a figura abaixo:



Fase Map

A fase do map aplica a mapfunction para toda a entrada do algoritmo. Para que isso aconteça, os mappers são executados em todos os nós computacionais do cluster, cuja tarefa é processar os blocos no arquivo de entrada que estão armazenados no HDFS. Estes blocos são divididos de forma mais igualitária possível e enviados para processamento localmente no nó do cluster. Em outras palavras, os cálculos ocorrem onde os dados são armazenados (localidade de dados).

Por não existir dependências entre os mappers, eles podem trabalhar em paralelo e independentemente um do outro. Se um computador nó do cluster falhar, os resultados do map serão desde nós serão recalculados em outro nó computacional que possui uma réplica do bloco correspondente. Um mapper processa o conteúdo do bloco linha a linha, interpretando cada linha como um par de chave-valor. A função de map real é chamada individualmente para cada um desses pares e cria uma lista arbitrariamente grande de novos pares de chave-valor.

Fase Shuffle

A fase Shuffle ordena os pares resultantes da fase do map localmente por suas chaves, depois disso, o MapReduce os atribui a um reducer de acordo com suas chaves. O framework garante que todos os pares com a mesma chave sejam atribuídos ao mesmo reducer. Como a saída da fase do map pode ser distribuída arbitrariamente em todo o cluster, a saída da fase do map precisa ser transferida pela rede para os produtores corretos na fase Shuffle. Por causa disso, é normal que grandes volumes de dados cruzem a rede nesta etapa.

Fase Reduce

Finalmente a fase de reduce coloca todos os pares com a mesma chave e cria uma lista classificada dos valores. A chave e a lista ordenada de valores fornecem a entrada para a função Reduce. A função Reduce normalmente compacta a lista de valores para criar uma lista mais curta - por exemplo, agregando os valores. Comumente, ele retorna um único valor como sua saída. De um modo geral, a função Reduce cria uma lista arbitrariamente grande de pares de chave-valor. A saída da fase de reduce pode, se necessário, ser usada como a entrada para outra iteração MapReduce.

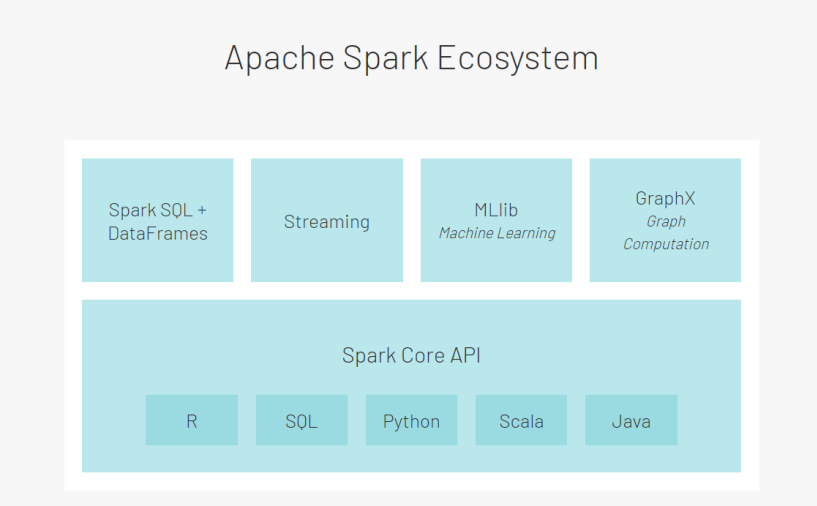
Para facilitar o entendimento buscamos um vídeo de exemplo de funcionamento do MapReduce: https://www.youtube.com/watch?v=mpCMUkf6WSQ

Spark

O Apache Spark é um framework de código aberto para Big Data que tem o objetivo de processar grandes conjuntos de dados de forma paralela e distribuída. Spark foi inicialmente construído por Matei Zaharia na Universidade da California (UC Berkeley) no AMPLab em 2009, e aberto em 2010 sob uma licença BSD. Em 2013, o projeto foi doado para a Apache Software Foundation e mudou sua licença para o Apache 2.0. Em fevereiro de 2014, Spark tornou-se um Projeto Apache Top-Level. Em novembro de 2014, a empresa Databricks do fundador do Spark, M. Zaharia, estabeleceu um novo recorde mundial na classificação em grande escala usando o Spark. Spark teve mais de 1.000 contribuidores em 2015, tornando-o um dos projetos mais ativos na Apache Software Foundation e um dos projetos de Big Data de código aberto mais ativos. No dia da construção desse material, março de 2021, o Apache Spark está com 1.628 contribuidores e está na versão 3.11 lançada em 02 de março de 2021[1].

O Apache Spark ou apenas Spark estende o modelo de programação MapReduce popularizado pelo Apache Hadoop, facilitando bastante o desenvolvimento de aplicações de processamento de grandes volumes de dados. Além disso, ele permite a programação nas linguagens: R, Java, Scala, SQL e Python.

O Spark tem diversos componentes para diferentes tipos de processamentos, todos construídos sobre o Spark Core, que é o componente que disponibiliza as funções básicas para o processamento como as funções `map`, `reduce`, `filter` e `collect`[2]. Entre estes destacam-se s presentes na Figura abaixo:



- O Spark Streamming: É o módulo que permite criar aplicações de *streaming* escaláveis e tolerante a falhas. Permite reutilizar o mesmo código para processamento em lote, unir os fluxos em relação aos dados históricos ou executar consultas iterativas ou em lote (batch). Permite ler dados do HDFS, Flume, Kafka, Twitter, entre outras fontes. Exemplo:

```
TwitterUtils.createStream(...)
    .filter(_.getText.contains("Spark"))
    .countByWindow(Seconds(5))
```

Contar tweets em um janela de tempo

- O SparkSQL + Dataframes: É o módulo utilizado para trabalhar com **dados estruturados**. Possibilita realizar consultas em dados estruturados dentro dos programas Spark, utilizando SQL ou a API do Spark para DataFrame. Possui compatibilidade com Metadados do Hive, amplamente utilizado em soluções Hadoop. A forma de conectividade padrão do Spark nas estruturas de dados ocorre por JDBC ou ODBC. Abaixo alguns exemplos de trechos de código SparkSQL:

```
context = HiveContext(sc)
results = context.sql(
  "SELECT * FROM people")
names = results.map(lambda p: p.name)
```

Aplicar funções ao resultado da consulta SQL

```
context.jsonFile("s3n://...")
  .registerTempTable("json")
results = context.sql(
  """SELECT *
     FROM people
     JOIN json ...""")
```

Consultar e juntar diferentes fontes de dados

- GraphX: É uma API para grafos e computação paralela de grafos. Compete com o desempenho dos sistemas existentes de grafo. Mantendo a flexibilidade do Spark, com tolerância a falhas e simples de usar, além de ter uma crescente biblioteca de algoritmos.

```
graph = Graph(vertices, edges)
messages = spark.textFile("hdfs://...")
graph2 = graph.joinVertices(messages) {
  (id, vertex, msg) => ...
}
```

Utilizando GraphX no Scala

- MLlib: É uma biblioteca de aprendizado de máquina escalável. Contém uma extensa lista de algoritmos de aprendizado de máquina como:
  - Classificação, regressão, árvores de decisão, sistema de recomendação, clusterização, entre outros.

  Também possibilita transformações de atributos, como categorização, normalização, entre outros.
- Spark-Core: Além dos componentes Streaming, SQL, ML e GraphX, precisamos falar do Spark-core. Spark-core é o mecanismo principal do Spark. Ele fornece serviços como gerenciamento de memória, agendamento de tarefas no cluster, recuperação de falhas, além de fornecer suporte para diversos sistemas de armazenamentos como HDFS, S3, entre outros. O Spark funciona como um sistema de processamento massivo em paralelo (MPP –Massively Parallel Processing) quando implantado em modo cluster e permite abstrair as camadas inferiores a trabalhar em um cluster através das APIs disponíveis.

```
data = spark.read.format("libsvm")\
  .load("hdfs://...")

model = KMeans(k=10).fit(data)
```

Exemplo MLlib Python

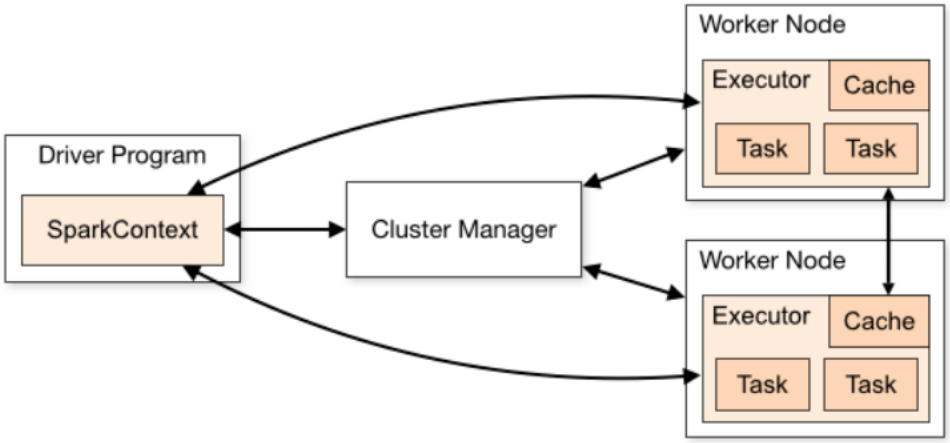
- Spark-Core: Além dos componentes Streaming, SQL, ML e GraphX, precisamos falar do Spark-core. Spark-core é o mecanismo principal do Spark. Ele fornece serviços como gerenciamento de memória, agendamento de tarefas no cluster, recuperação de falhas, além de fornecer suporte para diversos sistemas de armazenamentos como HDFS, S3, entre outros. O Spark funciona como um sistema de processamento massivo em paralelo (MPP –Massively Parallel Processing) quando implantado em modo cluster e permite abstrair as camadas inferiores a trabalhar em um cluster através das APIs disponíveis.

Arquitetura do Spark

Nessa seção serão explicadas as principais funcionalidades do Spark Core. Primeiro, será mostrada a arquitetura da solução e depois veremos os conceitos básicos no modelo de programação para o processamento de conjuntos de dados.

As aplicações Spark são executados como conjuntos independentes de processos em um cluster, coordenados pelo objeto **SparkContext** em seu programa principal (chamado de *driver program*). Para ser executado em um cluster, o SparkContext pode se conectar a vários tipos de gerenciadores de cluster (*cluster managers*) (seja o próprio gerenciador de cluster autônomo do Spark, Mesos ou YARN), que alocam recursos entre as aplicações. Uma vez conectado, o Spark adquire executores (*executors*) em nós no cluster, que são responsáveis por executar os processos e armazena os dados da aplicação. Em seguida, ele envia o código da aplicação (definido pelos arquivos JAR ou Python repassados ao SparkContext) para os executores. Finalmente, SparkContext envia tarefas para os executores rodarem.

A Figura 2 mostra a arquitetura do Spark e seus principais componentes.



Assim a arquitetura pode ser resumida pelas três partes principais:

- O **Driver Program**, que é a aplicação principal que gerencia a criação e é quem executará o processamento definido;
- O **Cluster Manager** é um componente opcional que só é necessário se o Spark for executado de forma distribuída. Ele é responsável por administrar as máquinas que serão utilizadas como workers;
- Os **Workers**, que são as máquinas que realmente executarão as tarefas que são enviadas pelo Driver Program. Se o Spark for executado de forma local, a máquina desempenhará tanto o papel de Driver Program como de Worker.

Existem várias coisas úteis a serem observadas sobre esta arquitetura:

1. Cada aplicação obtém seus próprios processos executores, que permanecem ativos durante toda a aplicação e executam tarefas (*tasks*) em vários threads. Isso tem a vantagem de isolar as aplicações uma da outra, tanto no lado do agendamento (cada driver agenda suas próprias tarefas) quanto no lado do executor (tarefas de diferentes aplicações executadas em diferentes JVMs). No entanto, também significa que os dados não podem ser compartilhados entre diferentes aplicatições Spark (instâncias do SparkContext) sem gravá-los em um sistema de armazenamento externo.
2. O Spark é agnóstico a qualquer gerenciador de cluster. Significa que diferentes gerenciadores de clusters podem ser utilizados para executar processos Spark. Contanto que ele possa adquirir processos do executor, e estes se comuniquem entre si, é relativamente fácil executá-lo, mesmo em um gerenciador de cluster que também oferece suporte a outras aplicações (por exemplo, Mesos / YARN).
3. O Driver Program deve escutar e aceitar conexões de entrada de seus executores ao longo de sua vida. Como tal, o Driver Program deve ser endereçável à rede a partir dos nós do Work.
4. Como o Driver Program agenda tarefas (*tasks*) no cluster, ele deve ser executado próximo aos nós Work, de preferência na mesma rede local.

Tipos de Cluster Manager

O Spark atualmente oferece suporte a vários gerenciadores de cluster:

- Standalone - um gerenciador de cluster simples incluído no Spark que facilita a configuração de um cluster. Normalmente usado em execuções locais
- Apache Mesos - um gerenciador geral de cluster que também pode executar Hadoop MapReduce.
- Hadoop YARN - o gerenciador de recursos no Hadoop 2.
- Kubernetes - um sistema de código aberto para automatizar a implantação, escalonamento e gerenciamento de aplicativos em contêineres.

Submetendo uma Aplicação

Aplicações podem ser submetidas a um cluster Spark através do script `spark-submit`. Mais informações em https://spark.apache.org/docs/latest/submitting-applications.html

Monitorando uma Aplicação

Cada Driver Program possui uma IU da web, normalmente na porta 4040, que exibe informações sobre a execução de tarefas, executores e uso de armazenamento. Basta ir para `http://driver-node:4040` em um navegador da web para acessar essa IU. O guia de monitoramento também descreve outras opções de monitoramento.

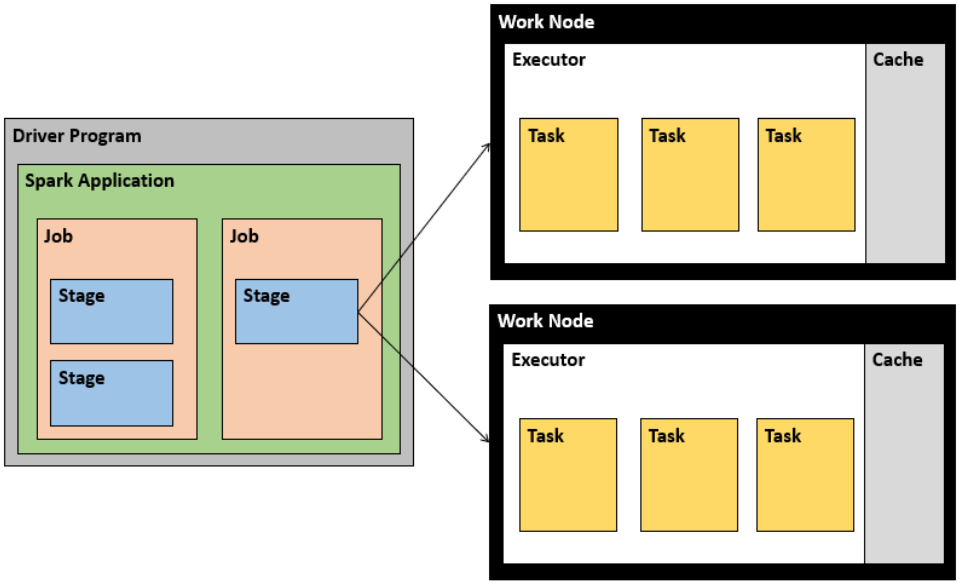
Glossário de Termos do Apache Spark

A tabela a seguir resume os termos que você verá neste material usados para se referir aos conceitos de cluster:

Termo	Significado
Application (Aplicação)	Programa do usuário desenvolvido em Spark. Consiste em um Driver Program e executores no cluster.
Application jar (Jar de Aplicação)	Um jar contendo uma aplicação Spark do usuário. Em alguns casos, os usuários desejarão criar um 'uber jar' contendo sua aplicação junto com suas dependências. O jar do usuário nunca deve incluir bibliotecas Hadoop ou Spark, no entanto, elas serão adicionadas no tempo de execução.
Driver Program	O processo que executa a função principal da aplicação e cria o SparkContext. Como se fosse o main() do programa.
Cluster manager	Um serviço externo para adquirir recursos no cluster (por exemplo, standalone, YARN)
Deploy mode	Distingue onde o processo do driver é executado. No modo 'cluster', o framework inicia o driver **dentro** do cluster. No modo 'cliente', o solicitante inicia o driver **fora** do cluster.
Worker node (Nó de Trabalho)	Qualquer nó que pode executar o código da aplicação no cluster
Executor	Um processo iniciado para uma aplicação Spark em um nó de trabalho, que executa tarefas e mantém os dados na memória ou armazenamento em disco. Cada aplicação possui seus próprios executores.
Task (Tarefa)	Uma tarefa ou unidade de trabalho que será enviada a um executor
Job (Trabalho)	Uma computação paralela que consiste em várias tarefas (Tasks) que são geradas em resposta a uma ação do Spark (por exemplo, `save`, `collect`)

Stage (Estágios)	Cada Job é dividido em conjuntos menores de tarefas chamados de estágios (Stages) que dependem uns dos outros (semelhante aos estágios map e reduzir no MapReduce)

Abaixo um exemplo de como os componentes ficam organizados:



Além da arquitetura, é importante conhecer os principais componentes do modelo de programação do Spark. O Spark tem uma arquitetura em camadas bem definida, com componentes fracamente acoplados, com base em duas abstrações principais:

- Resilient Distributed Datasets (RDDs)
- Directed Acyclic Graph (DAG)

Resilient Distributed Datasets

Os RDDs são essencialmente os blocos de construção do Spark - tudo é composto por eles. Mesmo as APIs de alto nível do Sparks (como por exemplo DataFrames) são compostas de RDDs subjacentes. O que significa ser um Resilient Distributed Dataset ou em português, Conjunto de Dados Distribuído Resiliente?

- **Resilient**: como o Spark é executado em um cluster de máquinas, a perda de dados por falha de hardware é uma preocupação muito real, então os RDDs são tolerantes a falhas e podem se reconstruir em caso de falha
- **Distributed**: um único RDD é armazenado em uma série de nós diferentes no cluster, não pertencendo a nenhuma fonte única (e nenhum ponto único de falha). Desta forma, o cluster pode operar o RDD em paralelo.
- **Dataset**: uma coleção de valores ou um conjunto de dados.

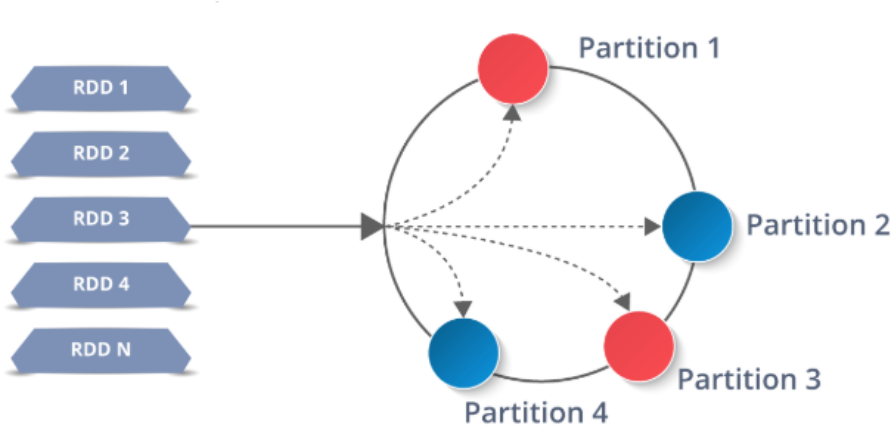
Todos os dados com os quais trabalhamos no Spark serão armazenados dentro de alguma forma de RDD - portanto, é fundamental compreendê-los totalmente.

O Spark oferece uma série de APIs de “Alto Nivel” construídas em cima de RDDs projetados para abstrair a complexidade de se criar RDDs, como DataFrames. De acordo com a

documentação do Apache Spark [3], um RDD é uma coleção de elementos particionados nos nós do cluster que podem ser operados em paralelo. Os RDDs são criados a partir de um arquivo no sistema de arquivos Hadoop (ou qualquer outro sistema de arquivos compatível com Hadoop) ou uma coleção Scala existente no driver program. O RDD ainda é fundamental para entender, pois é a estrutura subjacente de todos os dados no Spark.

Um RDD é coloquialmente equivalente a "Estrutura de dados distribuída". Um `JavaRDD<String>` é essencialmente apenas um `List<String>` disperso entre cada nó do cluster, com cada nó obtendo vários pedaços diferentes da lista. Com o Spark, precisa-se pensar em um contexto distribuído, sempre[7].

Os RDDs funcionam dividindo seus dados em uma série de partições a serem armazenadas em cada nó executor. Cada nó executará seu trabalho apenas em suas próprias partições. Isso é o que torna o Spark tão poderoso - se um executor morre ou uma tarefa falha, o Spark pode reconstruir apenas as partições de que precisa da fonte original e reenviar a tarefa para conclusão[7].



Os RDDs são imutáveis, o que significa que, uma vez criados, não podem ser alterados de forma alguma; eles só podem ser **transformados**. A noção de transformar RDDs está no cerne do Spark, e os jobs do Spark podem ser considerados nada mais do que qualquer combinação dessas etapas:

- Loading (carregar) dados para dentro de um RDD.
- Transforming (transformar) um RDD.
- Performing (executar) uma ação em um RDD.

O Spark define um conjunto de APIs para trabalhar com RDDs que podem ser divididos em dois grandes grupos - transformações e ações:

- Transformações: criam um novo conjunto de dados a partir de um existente
- Ações: retornam um valor ao driver program após a execução de um cálculo no conjunto de dados.

Por exemplo, **map** é uma transformação que passa cada elemento do conjunto de dados por meio de uma função e retorna um novo RDD que representa os resultados. Por outro lado, **reduce** é uma ação que agrega todos os elementos do RDD usando alguma função e retorna o

resultado final para o driver program (embora também haja um reduceByKey paralelo que retorna um conjunto de dados distribuído).

Por exemplo, a função de mapa `weatherData.map()` é uma transformação que passa cada elemento de um RDD por meio de uma função. Já o "reduce" é uma ação RDD que agrega todos os elementos de um RDD usando alguma função e retorna o resultado final para o driver program.

De acordo com a documentação do Spark[3], todas as transformações no Spark são preguiçosas, pois não calculam seus resultados imediatamente. Aí podemos lembrar aquela frase do Bill Gates: "I choose a lazy person to do a hard job. Because a lazy person will find an easy way to do it" = Eu escolho uma pessoa preguiçosa para fazer um trabalho difícil. Porque uma pessoa preguiçosa encontrará uma maneira fácil de fazer isso.

O que o Spark quer dizer com isso?

Que quando solicitamos ao Spark para criar um RDD por meio de transformações de um RDD existente, ele não gerará esse conjunto de dados até que uma ação específica seja realizada nele ou em um de seus filhos. O Spark executará a transformação e a ação que a desencadeou. Isso permite que o Spark funcione com muito mais eficiência. Por exemplo, podemos perceber que um conjunto de dados criado por meio do **map** será usado em um **reduce** e retornará apenas o resultado da redução para o driver, em vez do conjunto de dados maior mapeado no map.

RDD – Transformações:

Como vimos, as transformações moldam o conjunto de dados. **Elas só acontecem se uma ação for realizada**. Existem diversas transformações que podem ser realizadas, mas nesse material iremos testar apenas essas abaixo:

map, flatMap, mapValue, filter, reduceByKey, union, sortByKey, join e distinct.

map(func): Aplica a função (func) em cada elemento do RDD e retorna um RDD com o resultado.

```
In [1]: from pyspark import SparkContext

In [2]: sc = SparkContext("local", "aplicacao")

In [3]: rdd1 = sc.parallelize(["Clint Kakos", "Clint Noell", "John Gary Vargas", "Viola Davis", "John Kakos", "Clint Noell"])

In [4]: rdd1.map(lambda x: (x, 1)).collect()

Out[4]: [('Clint Kakos', 1),
         ('Clint Noell', 1),
         ('John Gary Vargas', 1),
         ('Viola Davis', 1),
         ('John Kakos', 1),
         ('Clint Noell', 1)]
```

flatMap(func): Similar ao map, porém cada entrada pode ser mapeada para 0 ou mais itens de saída (então a função deve retornar uma sequência ao invés de um único item).

```
In [5]: rdd1.flatMap(lambda item: item.split(' ')).collect()

Out[5]: ['Clint',
         'Kakos',
         'Clint',
         'Noell',
         'John',
         'Gary',
         'Vargas',
         'Viola',
         'Davis',
         'John',
         'Kakos',
         'Clint',
         'Noell']
```

mapValues(func): Aplica a função (func) em cada valor do par chave-valor (K, V) do RDD sem alterar a chave.

```
In [6]: rdd2 = sc.parallelize([("Clint", ["Futebol", "VideoGame", "Filmes"]), ("John", ["Filmes", "Pescar"])])

In [8]: def contar_hobbies(x):
            return len(x)

In [9]: rdd2.mapValues(contar_hobbies).collect()

Out[9]: [('Clint', 3), ('John', 2)]
```

filter(func): Retorna um RDD com apenas elementos que aceitam a condição do filtro.

```
In [10]: rdd1.filter(lambda x: x.startswith("C")).collect()

Out[10]: ['Clint Kakos', 'Clint Noell', 'Clint Noell']
```

reduceByKey(func): Unifica os valores para cada chave e retorna um conjunto de dados do par(K, V) onde os valores para cada chave são agregados utilizando a função de redução (func).

```
In [11]: def somar(x, y):
             return x + y

In [12]: rdd3 = rdd1.flatMap(lambda item: item.split(' ')).map(lambda x: (x, 1))

In [13]: rdd3.reduceByKey(somar).collect()

Out[13]: [('Clint', 3),
          ('Kakos', 2),
          ('Noell', 2),
          ('John', 2),
          ('Gary', 1),
          ('Vargas', 1),
          ('Viola', 1),
          ('Davis', 1)]
```

groupByKey(): Agrupa os valores de cada chave (K, V)do RDD em uma única sequência e retorna um conjunto de dados de par (K, Seq([V]).

```
In [14]: rdd3.groupByKey().collect()

Out[14]: [('Clint', <pyspark.resultiterable.ResultIterable at 0x20ebbe69908>),
          ('Kakos', <pyspark.resultiterable.ResultIterable at 0x20ebbe698d0>),
          ('Noell', <pyspark.resultiterable.ResultIterable at 0x20ebbe69a58>),
          ('John', <pyspark.resultiterable.ResultIterable at 0x20ebbe69978>),
          ('Gary', <pyspark.resultiterable.ResultIterable at 0x20ebbe69b38>),
          ('Vargas', <pyspark.resultiterable.ResultIterable at 0x20ebbe69ba8>),
          ('Viola', <pyspark.resultiterable.ResultIterable at 0x20ebbe69c18>),
          ('Davis', <pyspark.resultiterable.ResultIterable at 0x20ebbe69a90>)]

In [15]: rdd3.groupByKey().mapValues(list).collect()

Out[15]: [('Clint', [1, 1, 1]),
          ('Kakos', [1, 1]),
          ('Noell', [1, 1]),
          ('John', [1, 1]),
          ('Gary', [1]),
          ('Vargas', [1]),
          ('Viola', [1]),
          ('Davis', [1])]

In [16]: rdd3.groupByKey().mapValues(len).collect()

Out[16]: [('Clint', 3),
          ('Kakos', 2),
          ('Noell', 2),
          ('John', 2),
          ('Gary', 1),
          ('Vargas', 1),
          ('Viola', 1),
          ('Davis', 1)]
```

union(outroRDD): Realiza a união do RDD com outroRDD.

```
In [17]: rdd4 = sc.parallelize(["John Peter", "John Clint"])

In [18]: rdd1.union(rdd4).collect()

Out[18]: ['Clint Kakos',
          'Clint Noell',
          'John Gary Vargas',
          'Viola Davis',
          'John Kakos',
          'Clint Noell',
          'John Peter',
          'John Clint']
```

sortByKey(): Ordena o RDD, assumindo que consiste em pares (K, V).

```
In [19]: rdd3.sortByKey().collect()

Out[19]: [('Clint', 1),
          ('Clint', 1),
          ('Clint', 1),
          ('Davis', 1),
          ('Gary', 1),
          ('John', 1),
          ('John', 1),
          ('Kakos', 1),
          ('Kakos', 1),
          ('Noell', 1),
          ('Noell', 1),
          ('Vargas', 1),
          ('Viola', 1)]
```

join(outroRDD): Retorna um RDD contendo todos os pares de elementos que correspondem a chaves do RDD e do outroRDD.

```
In [20]: rdd5 = rdd3.reduceByKey(somar)

In [21]: rdd6 = sc.parallelize([("Noell", 1), ("Clint", 4), ("Peter", 20)])

In [22]: rdd5.join(rdd6).collect()

Out[22]: [('Noell', (2, 1)), ('Clint', (3, 4))]
```

distinct(): Retorna um novo RDD contendo os elementos distintos.

```
In [23]: rdd1.flatMap(lambda item: item.split(' ')).distinct().collect()

Out[23]: ['Clint', 'Kakos', 'Noell', 'John', 'Gary', 'Vargas', 'Viola', 'Davis']
```

Mais informações sobre Transformações de RDD em: http://spark.apache.org/docs/latest/rdd-programming-guide.html#transformations

RDD – Ações:

Ao contrário das transformações, as **Ações** executam as tarefas agendadas no conjunto do de dados e retornam algum valor. Existem diversas ações que podem ser realizadas, iremos testar:

reduce, collect, count, first, take, saveAsTextFile, countByKey

reduce(func): Agrega os elementos de um conjunto de dados utilizando a função (func) que deve receber dois argumentos e retornar um.

```
In [24]: def somar(x, y):
             return x+y

In [25]: rdd7 = rdd5.join(rdd6)

In [26]: rdd7.collect()
Out[26]: [('Noell', (2, 1)), ('Clint', (3, 4))]

In [27]: rdd7.flatMap(lambda valores: valores[1]).reduce(somar)
Out[27]: 10
```

collect(): Retorna todos os elementos de um conjunto de dados. Normalmente é utilizado depois da transformação filtrou outra operação que retorna um pequeno subconjunto dos dados.

```
In [26]: rdd7.collect()
Out[26]: [('Noell', (2, 1)), ('Clint', (3, 4))]
```

count(): Retorna o número de elementos em um conjunto de dados.

```
In [28]: rdd3.count()
Out[28]: 13
```

first(): Retorna o primeiro elemento no conjunto de dados.

```
In [29]: rdd3.first()
Out[29]: ('Clint', 1)
```

take(n): Retorna um array com o(s) primeiro(s) **n** elemento(s) do conjunto de dados.

```
In [30]: rdd5.take(3)
Out[30]: [('Clint', 3), ('Kakos', 2), ('Noell', 2)]

In [31]: rdd5.filter(lambda x: x[1] > 1).take(3)
Out[31]: [('Clint', 3), ('Kakos', 2), ('Noell', 2)]

In [38]: rdd5.filter(lambda x: x[1] > 1).take(2)
Out[38]: [('Clint', 3), ('Kakos', 2)]
```

countByKey(): Conta o número de elementos de cada chave e retorna o resultado para o programa principal como um dicionário.

```
In [39]: rdd3.countByKey()

Out[39]: defaultdict(int,
                     {'Clint': 3,
                      'Kakos': 2,
                      'Noell': 2,
                      'John': 2,
                      'Gary': 1,
                      'Vargas': 1,
                      'Viola': 1,
                      'Davis': 1})
```

saveAsTextFile(path): Escreve os elementos do conjunto de dados para um arquivo de texto dentro de um diretório no sistema de arquivo local, HDFS, S3 entre outros.

```
In [40]: rdd5.saveAsTextFile('Arquivo_teste')
```

Mais informações sobre Ações de RDD em: http://spark.apache.org/docs/latest/rdd-programming-guide.html#actions

Directed Acyclic Graph

Sempre que uma ação é executada em um RDD, o Spark cria um DAG - um grafo direto finito sem ciclos direcionados (caso contrário, nosso trabalho seria executado para sempre).

Lembre-se de que um grafo nada mais é do que uma série de vértices e arestas conectadas, e este grafo não é diferente. Cada vértice no DAG é uma função Spark; alguma operação realizada em um RDD (`map, mapToPair, reduceByKey`, etc).

No `MapReduce`, o DAG consiste em dois vértices: `Map` → `Reduce`.

O DAG permite que o Spark otimize seu plano de execução e minimize o shuffling (fase de ordenação do MapReduce).

Até agora usamos o Spark Shell para executar comandos Spark na linguagem Python. O Spark Shell fornece uma maneira simples de executar um código Spark em Scala ou Python, e é apenas uma das maneiras de executar programas Spark. Antes vamos entender a estrutura do Spark que instalamos.

Estrutura de diretórios do Spark

Dentro da instalação do Spark, a estrutura de diretórios tem a seguinte finalidade:

- bin–Contém os arquivos executáveis para iniciar o shell interativo (Scala, Python e R), bem como o spark-submit que é utilizado para submeter uma aplicação para ser executada no Spark.
- conf–Contém os arquivos de configuração para definir o formato do log, as métricas, quais são os nós trabalhadores (slaves), além do arquivo que contém as variáveis de ambiente do Spark.
- sbin–Contém diversos scripts que auxiliam na execução e configuração de um cluster.
- data–Contém alguns exemplos de conjunto de dados para as bibliotecas MLLib, Streaming e GraphX. Esses conjuntos de dados possibilitam executar os exemplos disponíveis na documentação Spark.
- jars–Contém diversos arquivos .jarcom diferentes classes que possibilitam ao Spark realizar as computações necessárias.

- examples–Contém o código e arquivos .jar de exemplos escritos nas linguagens suportadas pelo Spark (Java, Scala, Python e R). Nesta pasta também contém alguns conjuntos de dados.

Código Spark

A primeira parte do código Spark é definida pelo SparkSession ou SparkContext. O **SparkContext** é um ponto de entrada para Spark e é definido no pacote `org.apache.spark` desde a versão 1.x. É usado para criar programas Spark com RDD, acumuladores e variáveis de transmissão no cluster. Seu objeto `sc` está disponível por padrão no spark-shell e pode ser criado programaticamente usando a classe SparkContext. Podemos usar esse objeto diretamente quando necessário

```
val rdd = sc.textFile("/src/main/resources/text/alice.txt")
```

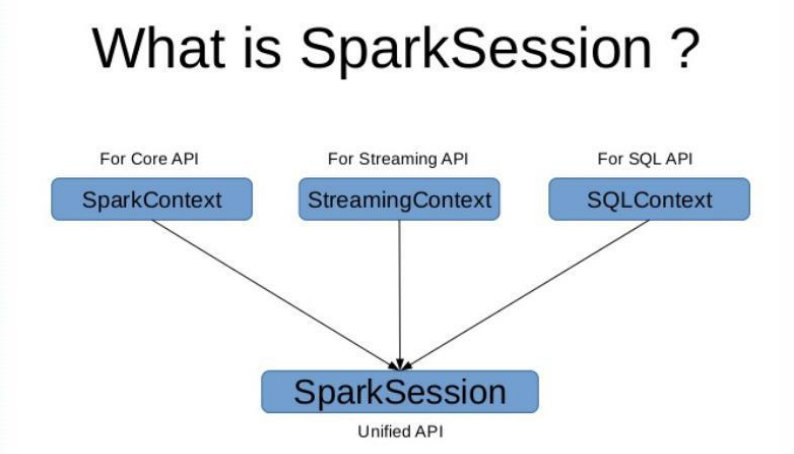
Desde o Spark 2.x, a maioria das funcionalidades (métodos) disponíveis no SparkContext também estão disponíveis no **SparkSession**.

Com o Spark 2.x, uma nova classe `org.apache.spark.sql.SparkSession` foi introduzida. Na qual é uma classe combinada para todos os contextos diferentes como SQLContext, StreamingContext, etc. Assim, o **SparkSession** é a primeira parte do código Spark na qual cria uma instância SparkSession. Deve ser a primeira instrução que você escreve para programar com RDD, DataFrame e Dataset.

O Spark Session também inclui todas as APIs disponíveis em diferentes contextos:

- Spark Context
- SQL Context
- Streaming Context
- Hive Context*

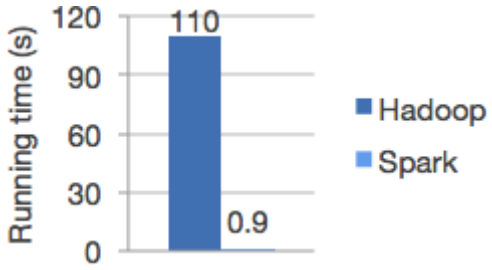
*Obs.: O SQLContext e HiveContext forma unificados em um objeto a partir do Spark 2.



Spark vs Hadoop

Além do modelo de programação estendido, o Spark também apresenta uma performance muito superior ao Hadoop, chegando em alguns casos a apresentar uma performance quase 100x maior. Outra grande vantagem do Spark, é que todos os componentes funcionam integrados na própria ferramenta, como o Spark Streamming, o Spark SQL e o GraphX, diferentemente do Hadoop, onde é necessário utilizar ferramentas que se integram a ele, mas que são distribuídas separadamente, como o Apache Hive[2]

De acordo com a Apache[3], O Apache Spark atinge alto desempenho para dados em batch e streaming, usando um agendador DAG de última geração, um otimizador de consulta e um mecanismo de execução física. Assim conseguindo resultados muito bons, até 100x mais rápido que no Hadoop nas versões 2. No exemplo abaixo de uma Regressão Logística:



De acordo com o Release notes da versão 3 do Spark, o Spark 3.0 é aproximadamente duas vezes mais rápido que o Spark 2.4. E agora a linguagem Python é a linguagem mais usada no Spark.

Principais diferenças entre Hadoop e Spark

Para listar as principais diferenças usamos o comparativo feito pelo PhoenixNap [8]. As seções a seguir descrevem as principais diferenças e semelhanças entre as duas estruturas. Vamos dar uma olhada no Hadoop vs. Spark de vários ângulos.

Hadoop	Categoria de Comparação	Spark
Desempenho mais lento, usa discos para armazenamento e depende da velocidade de leitura e gravação do disco.	**Performance**	Desempenho rápido na memória com operações reduzidas de leitura e gravação em disco.
Uma plataforma de Open-Source menos dispendiosa para operar. Usa hardware de consumo acessível	**Custo**	Uma plataforma de Open-Source, mas depende da memória para computação, o que aumenta consideravelmente os custos de operação.
Melhor para processamento em batch. Usa MapReduce para dividir um grande conjunto de	**Processamento de Dados**	Adequado para análise de dados iterativa e analise em em near real-time. Funciona com

dados em um cluster para análise paralela.		RDDs e DAGs para executar operações.
Um sistema altamente tolerante a falhas. Replica os dados entre os nós e os usa no caso de um problema.	**Tolerância a Falhas**	Rastreia o processo de criação do bloco RDD e pode reconstruir um conjunto de dados quando uma partição falha. O Spark também pode usar um DAG para reconstruir dados entre nós.
Facilmente escalável adicionando nós ao cluster e discos para armazenamento. Suporta dezenas de milhares de nós sem um limite conhecido.	**Escalabilidade**	Um pouco mais desafiador de escalar porque depende de RAM para computação. Suporta milhares de nós em um cluster.
Extremamente seguro. Suporta LDAP, ACLs, Kerberos, SLAs, etc.	**Segurança**	Não é seguro. Por padrão, a segurança está desligada. Depende da integração com o Hadoop para atingir o nível de segurança necessário.
Mais difícil de usar com menos linguagens suportados. Usa Java ou Python para aplicações MapReduce.	**Facilidade de Uso e Linguagens Suportadas**	Mais amigável. Permite o modo de shell interativo. As APIs podem ser escritas em Java, Scala, R, Python, Spark SQL.
Mais lento que o Spark. Os fragmentos de dados podem ser muito grandes e criar gargalos. Mahout é a biblioteca principal.	**Machine Learning**	Muito mais rápido com processamento na memória. Usa a biblioteca MLlib.
Usa soluções externas. YARN é a opção mais comum para gerenciamento de recursos. O Oozie está disponível para agendamento de fluxo de trabalho.	**Gerenciamento de Recursos e Agendamento**	Possui ferramentas integradas para alocação, programação e monitoramento de recursos.

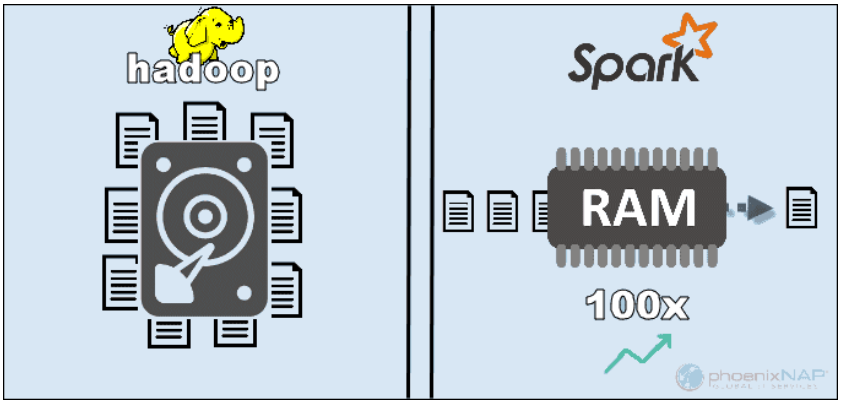
Alguns deles são custo, desempenho, segurança e facilidade de uso.

Performance

Quando observamos Hadoop vs. Spark em termos de como eles processam dados, pode não parecer natural comparar o desempenho dos dois frameworks. Ainda assim, podemos traçar uma linha e obter uma imagem clara de qual ferramenta é mais rápida. Acessando os dados armazenados localmente no HDFS, o Hadoop aumenta o desempenho geral. No entanto, não é páreo para o processamento na memória do Spark. De acordo com as afirmações da Apache, o Spark parece ser 100x mais rápido ao usar RAM para computação do que o Hadoop com MapReduce.

O domínio permaneceu com a classificação dos dados em discos. O Spark era 3x mais rápido e precisava de 10x menos nós para processar 100 TB de dados no HDFS. Essa referência foi suficiente para estabelecer o recorde mundial em 2014. A principal razão para essa supremacia do Spark é que ele não lê e grava dados intermediários em discos, mas usa RAM. O Hadoop

armazena dados em muitas fontes diferentes e, em seguida, processa os dados em batches usando MapReduce.



Todos os itens acima podem posicionar o Spark como o vencedor absoluto. No entanto, se o tamanho dos dados for maior do que a RAM disponível, o Hadoop é a escolha mais lógica. Outro ponto a ser considerado é o custo de funcionamento desses sistemas.

Custo

Comparando o Hadoop ao Spark com relação ao custo, precisamos ir mais fundo do que o preço do software. Ambas as plataformas são de código aberto e totalmente gratuitas. No entanto, os custos de infraestrutura, manutenção e desenvolvimento precisam ser levados em consideração para obter um aproximado Custo Total de Propriedade, do inglês Total Cost of Ownership (TCO).

O fator mais significativo na categoria de custo é o hardware subjacente de que você precisa para executar essas ferramentas. Como o Hadoop depende de qualquer tipo de armazenamento em disco para processamento de dados, o custo de execução é relativamente baixo. Por outro lado, o Spark depende de cálculos na memória para processamento de dados em tempo real. Portanto, a rotação de nós do cluster com muita RAM aumenta consideravelmente o custo de propriedade.

Os pontos acima sugerem que a infraestrutura do Hadoop é mais econômica. Embora essa afirmação esteja correta, precisamos lembrar de que o Spark processa dados muito mais rápido. Portanto, requer um número menor de máquinas para concluir a mesma tarefa.

Processamento de Dados

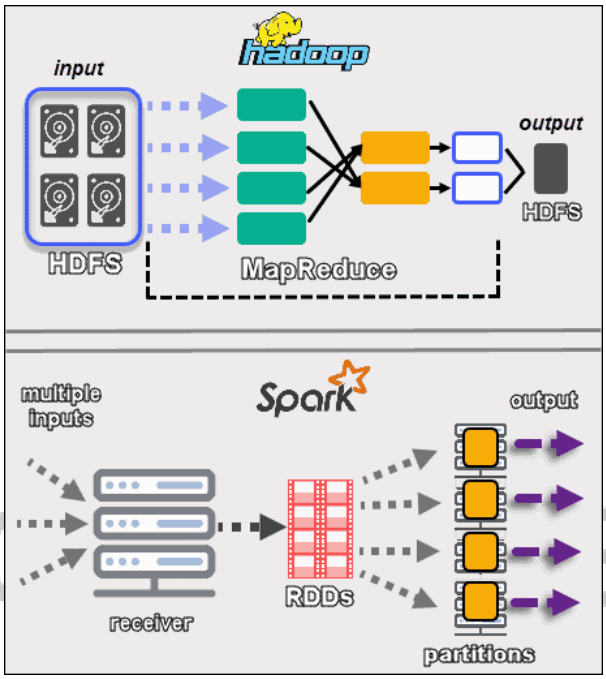
As duas estruturas lidam com dados de maneiras bastante diferentes. Embora o Hadoop com MapReduce e o Spark com RDDs processem dados em um ambiente distribuído, o Hadoop é mais adequado para processamento em batch. Em contraste, o Spark brilha com processamento em tempo real.

O objetivo do Hadoop é armazenar dados em discos e, em seguida, analisá-los em paralelo em batches em um ambiente distribuído. O MapReduce não requer uma grande quantidade de RAM para lidar com grandes volumes de dados. O Hadoop depende de hardware diário para armazenamento e é mais adequado para processamento de dados linear.

O Apache Spark funciona com Resilient Distributed Datasets (RDDs). Um RDD é um conjunto distribuído de elementos armazenados em partições em nós do cluster. O tamanho de um RDD

geralmente é muito grande para um nó manipular. Portanto, o Spark particiona os RDDs para os nós mais próximos e executa as operações em paralelo. O sistema rastreia todas as ações realizadas em um RDD pelo uso de um Directed Acyclic Graph (DAG).

Com computação em memória e APIs de alto nível, o Spark lida de maneira eficaz com streams real time de dados não estruturados. Além disso, os dados são armazenados em um número predefinido de partições. Um nó pode ter quantas partições forem necessárias, mas uma partição não pode se expandir para outro nó.



Tolerância a Falhas

Falando de Hadoop vs. Spark na categoria de tolerância a falhas, podemos dizer que ambos fornecem um nível respeitável de tratamento de falhas. Além disso, podemos dizer que a maneira como eles abordam a tolerância a falhas é diferente.

O Hadoop tem tolerância a falhas como base de sua operação. Ele replica dados muitas vezes nos nós do cluster. Caso ocorra um problema, o sistema retoma o trabalho criando os blocos que faltam em outros locais. Os nós mestres rastreiam o status de todos os nós escravos. Finalmente, se um nó escravo não responder aos pings de um mestre, o mestre atribui os trabalhos pendentes a outro nó escravo.

O Spark usa blocos RDD para alcançar tolerância a falhas. O sistema rastreia como o conjunto de dados imutável é criado. Então, ele pode reiniciar o processo quando houver um problema. O Spark pode reconstruir dados em um cluster usando o rastreamento DAG dos fluxos de trabalho. Essa estrutura de dados permite que o Spark lide com falhas em um ecossistema de processamento de dados distribuído.

Escalabilidade

A linha entre o Hadoop e o Spark fica embaçada nesta seção. O Hadoop usa HDFS para lidar com big data. Quando o volume de dados cresce rapidamente, o Hadoop pode escalar

rapidamente para acomodar a demanda. Como o Spark não tem seu sistema de arquivos, ele precisa confiar no HDFS quando os dados são muito grandes para serem manipulados.

Os clusters podem facilmente expandir e aumentar o poder de computação adicionando mais servidores à rede. Como resultado, o número de nós em ambas as estruturas pode chegar a milhares. Não há limite para quantos servidores você pode adicionar a cada cluster e quantos dados você pode processar.

Alguns dos números confirmados incluem 8.000 máquinas em um ambiente Spark com petabytes de dados. Quando se fala em clusters Hadoop, eles são bem conhecidos por acomodar dezenas de milhares de máquinas e quase um exabyte de dados.

Facilidade de Uso e Linguagens Suportadas

O Spark fornece suporte para várias linguagens ao lado do idioma nativo (Scala): Java, Python, R e Spark SQL. Isso permite que os desenvolvedores usem a linguagem de programação de sua preferência.

A estrutura do Hadoop é baseada em Java. As duas principais linguagens para escrever código MapReduce são Java ou Python. O Hadoop não possui um modo interativo para auxiliar os usuários. No entanto, ele se integra às ferramentas Pig e Hive para facilitar a escrita de programas MapReduce complexos.

Além do suporte para APIs em várias linguagens, o Spark ganha na seção de facilidade de uso com seu modo interativo. Você pode usar o shell do Spark para analisar dados interativamente com Scala ou Python. O shell fornece feedback instantâneo para consultas, o que torna o Spark mais fácil de usar do que o Hadoop MapReduce.

Outra coisa que dá ao Spark vantagem é que os programadores podem reutilizar o código existente quando aplicável. Ao fazer isso, os desenvolvedores podem reduzir o tempo de desenvolvimento de aplicativos. Os dados históricos e de fluxo podem ser combinados para tornar esse processo ainda mais eficaz.

Segurança

Comparando o Hadoop com a segurança do Spark, vamos deixar o gato fora da bolsa imediatamente - o Hadoop é o vencedor absoluto. Acima de tudo, a segurança do Spark está desativada por padrão. Isso significa que sua configuração será exposta se você não resolver esse problema.

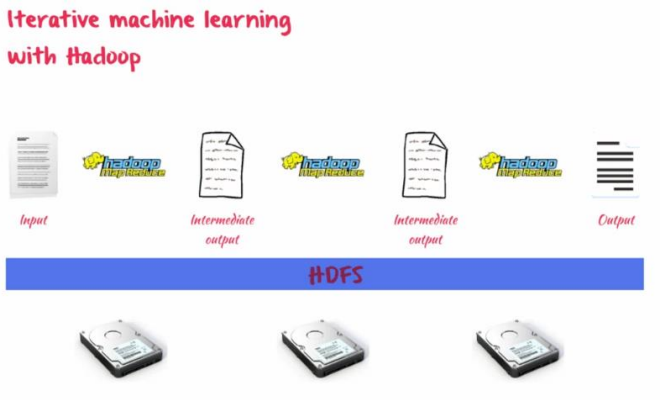
No entanto, o Spark pode atingir um nível adequado de segurança integrando-se ao Hadoop. Dessa forma, o Spark pode usar todos os métodos disponíveis para Hadoop e HDFS. Além disso, quando o Spark é executado no YARN, você pode adotar os benefícios de outros métodos de autenticação.

Machine Learning

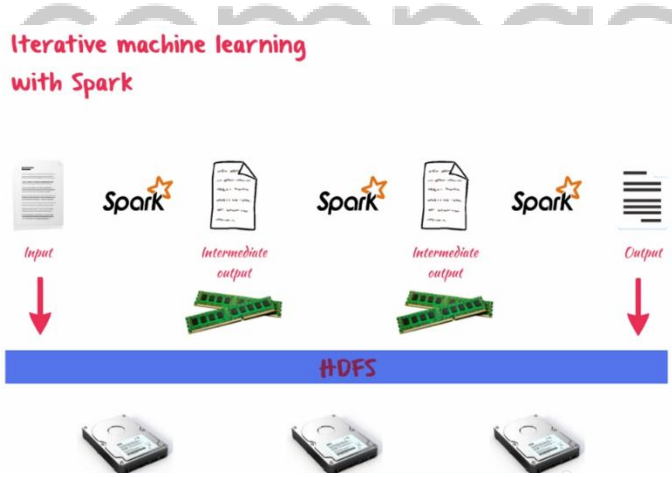
Machine Learning, ou em português aprendizado de máquina, é um processo iterativo que funciona melhor usando a computação in-memory. Por esse motivo, o Spark provou ser uma solução mais rápida nessa área.

A razão para isso é que o Hadoop MapReduce divide os trabalhos em tarefas paralelas que podem ser muito grandes para algoritmos de aprendizado de máquina. Esse processo cria problemas de desempenho de E/S nesses aplicativos Hadoop.

A biblioteca Mahout é a principal plataforma de aprendizado de máquina em clusters do Hadoop. Mahout depende do MapReduce para realizar clustering, classificação e recomendação.



O Spark vem com uma biblioteca de aprendizado de máquina padrão, MLlib. Esta biblioteca realiza computação iterativas de ML na memória. Inclui ferramentas para realizar regressão, classificação, persistência, construção de pipeline, avaliação e muito mais.



O Spark com MLlib provou ser nove vezes (9x) mais rápido do que o Apache Mahout em um ambiente baseado em disco Hadoop. Quando você precisa de resultados mais eficientes do que o que o Hadoop oferece, o Spark é a melhor escolha para aprendizado de máquina.

Gerenciamento de Recursos e Agendamento

O Hadoop não possui um agendador integrado. Ele usa soluções externas para gerenciamento de recursos e agendamento. Juntamente com ResourceManager e NodeManager, YARN é responsável pelo gerenciamento de recursos em um cluster Hadoop. Uma das ferramentas disponíveis para agendar fluxos de trabalho é o Oozie.

O YARN não lida com gerenciamento de estado de aplicações individuais. Ele apenas aloca o poder de processamento disponível.

O Hadoop MapReduce funciona com plug-ins como CapacityScheduler e FairScheduler. Esses agendadores garantem que as aplicações obtenham os recursos essenciais conforme necessário, mantendo a eficiência de um cluster. O FairScheduler fornece os recursos necessários para as aplicações enquanto mantém o controle de que, no final, todos as aplicações obtêm a mesma alocação de recursos.

O Spark, por outro lado, tem essas funções integradas. O DAG scheduler é responsável por dividir os operadores em estágios. Cada estágio tem várias tarefas que o DAG programa e o Spark precisa executar.

O Spark Scheduler e o Block Manager executam o agendamento de tarefas e tarefas, monitoramento e distribuição de recursos em um cluster.

A principal diferença entre o Hadoop MapReduce e o Spark

De fato, a principal diferença entre eles está na abordagem do processamento: o Spark pode fazer isso na memória, enquanto o Hadoop MapReduce precisa ler e gravar em um disco. Como resultado, a velocidade de processamento difere significativamente. O Spark pode ser até 100 vezes mais rápido. No entanto, o volume de dados processados também difere: o Hadoop MapReduce é capaz de trabalhar com conjuntos de dados muito maiores do que o Spark[9].

O Hadoop MapReduce é bom para:

- **Processamento linear de grandes conjuntos de dados.** O Hadoop MapReduce permite o processamento paralelo de grandes quantidades de dados. Ele divide um grande fragmento em partes menores para serem processadas separadamente em diferentes nós de dados e reúne automaticamente os resultados nos vários nós para retornar um único resultado. Caso o conjunto de dados resultante seja maior que a RAM disponível, o Hadoop MapReduce pode superar o Spark.
- **Solução econômica, se não houver resultados imediatos**. O Hadoop considera o MapReduce uma boa solução se a velocidade de processamento não for crítica. Por exemplo, se o processamento de dados puder ser feito durante a noite, faz sentido considerar o uso do MapReduce do Hadoop.

Spark é bom para:

- **Processamento rápido de dados**. O processamento na memória torna o Spark mais rápido que o Hadoop MapReduce — até 100 vezes para dados na RAM e até 10 vezes para dados armazenados.
- **Processamento iterativo**. Se a tarefa for processar dados de novo e de novo, o Spark elimina o Hadoop MapReduce. Os RDDs (Distributed Datasets) resilientes do Spark permitem várias operações de mapa na memória, enquanto o Hadoop MapReduce tem que gravar resultados provisórios em um disco.
- **Processamento quase em tempo real**. Se uma empresa precisar de insights imediatos, deverá optar pelo Spark e pelo processamento na memória.
- **Processamento gráfico**. O modelo computacional do Spark é bom para cálculos iterativos típicos no processamento de gráficos. E o Apache Spark tem o GraphX - uma API para computação gráfica.

- **Aprendizado de máquina.** O Spark possui MLlib — uma biblioteca de aprendizado de máquina integrada, enquanto o Hadoop precisa de um terceiro para fornecê-lo. O MLlib possui algoritmos prontos que também são executados na memória.
- **Juntando conjuntos de dados**. Devido à sua velocidade, o Spark pode criar todas as combinações mais rapidamente, embora o Hadoop possa ser melhor se for necessário juntar conjuntos de dados muito grandes que requeiram muito embaralhamento e classificação.

Exercícios:

1) Instale a última versão do Apache Spark em Python via Pip - https://spark.apache.org/docs/latest/api/python/getting_started/install.html ou https://www.datacamp.com/community/tutorials/installation-of-pyspark ou seguindo o Laboratório “Instalando e configurando Apache Spark”
2) Usando o Spark Shell faça um programa que conte as palavras de um arquivo README.md da pasta home do Spark – Somente da Seção Basics do link https://spark.apache.org/docs/latest/quick-start.html

Referências

[1] https://en.wikipedia.org/wiki/Apache_Spark

[2] https://www.devmedia.com.br/introducao-ao-apache-spark/34178

[3] https://spark.apache.org/

[4] https://www.devmedia.com.br/hadoop-fundamentos-e-instalacao/29466

[5] https://repositorio.unb.br/bitstream/10482/33759/1/2018_MarcoAnt%C3%B4niodeSousaReis.pdf

[6] https://www.splunk.com/pdfs/solution-guides/discover-hadoop.pdf

[7] https://betterprogramming.pub/high-level-overview-of-apache-spark-c225a0a162e9

[8] https://phoenixnap.com/kb/hadoop-vs-spark

[9] https://medium.com/mangue-data/spark-vs-hadoop-mapreduce-qual-estrutura-de-big-data-escolher-b8927de07f7e