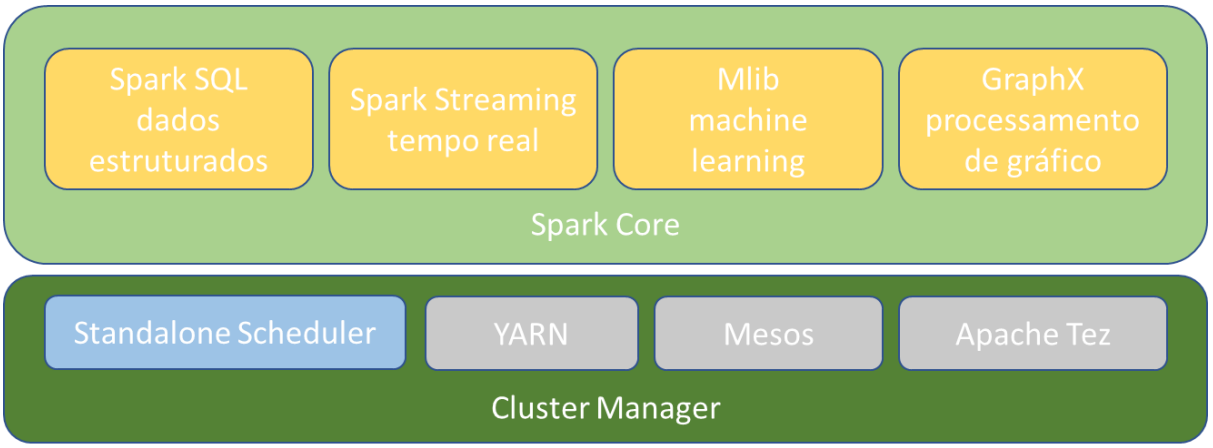
Tabela de conteúdo

eder.sferreira@gmail.com

Componentes do Spark



Spark Core: Contém a funcionalidade básica do Spark incluindo componentes para Agendamento de Tarefas, Gerenciamento de Memória, Recuperação de Falhas, fazendo interface com o sistema de armazenamento de dados entre outras funções. É nele que reside a API que define o RDD (*Resilient Distributed Dataset*) que é a principal abstração de programação Spark.

Spark SQL: É um pacote do Spark utilizado para trabalhar com dados estruturados. Ele permite consultar dados via SQL bem como o HQL do HIVE mas também suporta muitas fontes de dados como Parquet e JSON.

Spark Streaming: É um componente que possibilita transmissão ou tráfego de dados online e ao vivo. Como exemplo de dados de streaming estão os logs gerados por servidores web de produção ou filas de mensagens postados por usuários em webservice.

MLib: O Spark vem com uma funcionalidade comum de Machine Learn (ML) chamada MLib. Esse pacote fornece vários tipos de algoritmos de *machine learn* como classificação, regressão, clusterização e filtro colaborativo bem como suporta funcionalidades como evolução de modelo e importação de dados. Todas essas as funcionalidades são projetadas para serem escaladas horizontalmente em um cluster.

GraphX: Essa é uma biblioteca de manipulação de gráficos e execução de cálculos em paralelo de gráficos.

Cluster Managers: Essa “máquina” (Spark) é projetada para ser eficientemente escalável horizontalmente para vários milhares de nodes em um cluster. Para conseguir isso e maximizar a flexibilidade, o Spark pode ser executado em vários gerenciadores de cluster, incluindo *Hadoop YARN, Apache Tez, Apache Mesos* e um gerenciador de cluster simples incluído no próprio Spark chamado *Standalone Scheduler*. Se você estiver instalando o Spark em uma máquina, sem cluster, o *Standalone Scheduler* oferece uma maneira fácil de começar; se você já tiver um cluster Hadoop YARN ou Apache Tez, no entanto, o suporte do Spark para esses gerenciadores de cluster permite que seus aplicativos também sejam executados neles. Todos esses gerenciadores de cluster são plugáveis, ou seja, podem ser substituídos no Spark.

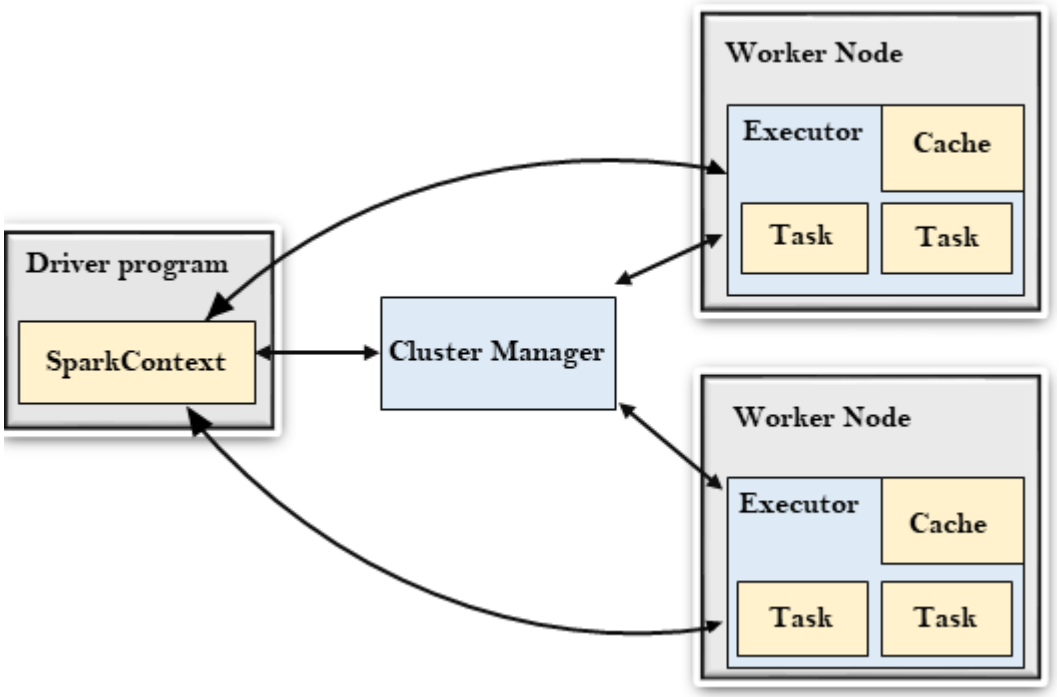
Standalone: É um gerenciador de cluster simples que é mais fácil de configurar e executar as tarefas no cluster. É um gerenciador de cluster confiável que pode facilmente gerenciar falhas. Ele pode gerenciar os recursos com base nos requisitos da aplicação submetida.

Apache Mesos: É um gerenciador geral de cluster do grupo Apache que também pode executar o Hadoop MapReduce junto com o Spark e outros aplicativos de serviço. Consiste em API para a maioria das linguagens de programação.

Hadoop YARN: É um gerenciador de recursos que foi fornecido no Hadoop 2. Significa *Yet Another Resource Negotiator*. Também é um gerenciador de cluster de uso geral e pode funcionar no Hadoop e no Spark.

Arquitetura do Spark

O spark consiste no modelo master-slave, ou seja, o cluster tem um master e um ou vários slaves. Vamos entender melhor como o Spark funciona olhando para os processos: **Driver program, Worker Node, Cluster Manager, Executor, Cache** e **Task.**



Driver Program: É um processo que executa a função *main()* da aplicação e cria o objeto *SparkContext*. O objetivo do *SparkContext* é coordenar os aplicativos spark, executando como conjuntos independentes de processos em um cluster.

Para executar em um cluster, o *SparkContext* se conecta a um gerenciador de cluster e, em seguida, executa as seguintes tarefas:

- Adquire *executors* em nodes no cluster.
- Em seguida, ele envia o código do aplicativo para os *executors*. O código da aplicação pode ser definido por arquivos JAR ou Python passados para o SparkContext.
- Por fim, o *SparkContext* envia tarefas para os *executors* trabalharem.

Worker Node: O Worker Node não é um processo e sim o próprio node do cluster que trabalha como slave. Sua função é executar o código da aplicação no cluster.

Executor: Este é um processo iniciado para uma aplicação em um *worker node.* Ele executa tarefas e mantém os dados na memória ou armazenamento em disco entre eles. Ele lê e grava dados nas fontes externas e cada aplicativo contém seu executor.

Tarefa: Uma unidade de trabalho que será enviada a um executor.

eder.sferreira@gmail.com

Exemplo de programa

Vamos utilizar o código abaixo como exemplo:

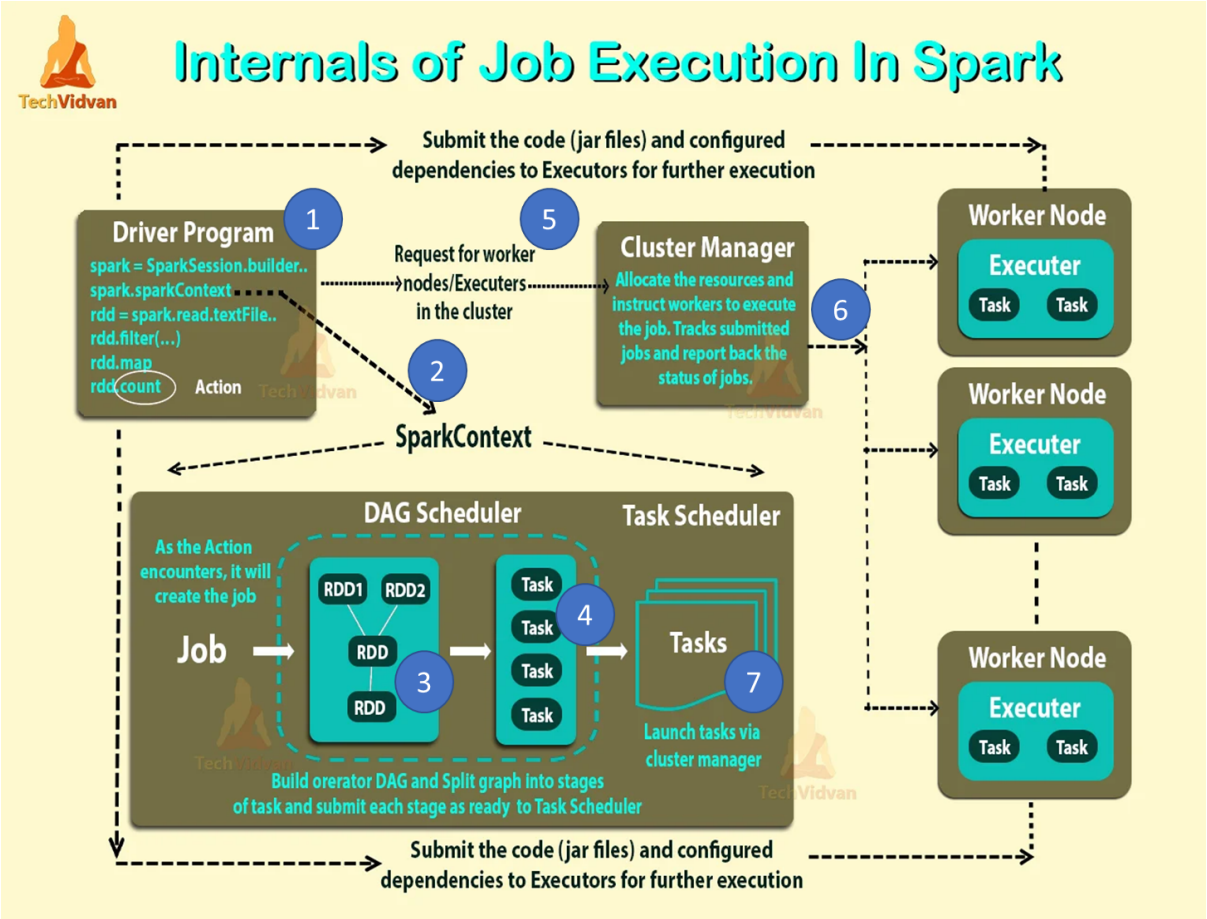
```
from pyspark import SparkContext
sc = SparkContext("local", "count app")
words = sc.parallelize (
   ["scala",
   "java",
   "hadoop",
   "spark",
   "akka",
   "spark vs hadoop",
   "pyspark",
   "pyspark and spark"]
)
words_filter = words.filter(lambda x: 'spark' in x)
counts = words_filter.count()
print "Number of elements filtered  -> %i" % (counts)
```

```
//sample output
Number of elements
filtered -> 4
```

Primeiro, estamos importando o *SparkContext* que é o ponto de entrada da funcionalidade do *Spark*. A segunda linha cria um objeto *SparkContext*. Em seguida, estamos criando uma amostra de 'palavras' RDD pelo método *parallelize*. As palavras que contêm a string ‘spark’ são filtradas e armazenadas em *words_filter*. A função *count()* é usada para contar o número de palavras filtradas e o resultado é impresso na tela.

Aqui, o processo de aplicar um filtro aos dados no RDD é **transformação** e contar o número de palavras é **ação**.

Agora, vamos analisar o que acontece quando um trabalho é enviado ao Spark.



Na imagem acima é descrito o fluxo de execução, mas vamos detalhar e entender a sequência desse trabalho.

1. A aplicação é enviada para o Driver que analisa o código e, quando encontra uma ação,

2. converte o código em um Gráfico Acíclico Direcionado (DAG). Depois, o driver executa certas otimizações, como transformações de pipeline.

3. Além disso, converte o DAG em plano de execução física com o conjunto de etapas. Enquanto isso,

4. ele cria pequenas unidades de execução em cada estágio, chamadas de tarefas. Em seguida,

5. conversa com o gerenciador do cluster e solicita alocação de recursos.

6. Neste ponto, o gerenciador de cluster negocia os recursos com os nodes. Os executores se registram no programa do Driver antes que eles iniciem a execução para que o Driver tenha a visão holística de todos os executores.

7. Agora, o Driver coleta todas as tarefas e as envia para o cluster através do Cluster Manager.

Uma vez que todas as tarefas foram enviadas para os executores, eles iniciam a execução e, enquanto a aplicação está rodando, o Driver monitora os executores.

RDD (Resilient Distributed Dataset)

O *Resilient Distributed Dataset* (RDD) forma a espinha dorsal do Apache Spark.

Resilient: Se o processo falhar, o dado é recuperado.
Distributed: Os dados são distribuídos entre todos os nodes.
Dataset: Grupo de dados

Um RDD é uma representação lógica de um dataset que pode ser **particionado** entre os nós do cluster. Como exemplo, veja o código abaixo:

```
val rdd = sc.textFile("/arquivo.txt",5)
val lines = sc.parallelize(List("Brasil","Japao","Australia"))
```

Neste exemplo, no primeiro comando, o segundo argumento com valor 5 determina o número de partições.

Como um RDD é uma coleção de vários dados, caso não caiba em um único nó ele deve ser particionado em vários nós, ou seja, quanto maior o número de partições, maior o paralelismo. Essas partições de um RDD são distribuídas por todos os nós da rede.

No Spark, há duas **operações** principais chamadas **transformação** e **ação**:

> **Transformações** criam um novo RDD com base nas operações que realizamos nos dados de entrada. Exemplos são *Map()*, *Filter()* e *sortBy()* que podemos realizar nos dados. Esse tipo de operação é **lazy**, ou seja, só é executada quando algum dado é requisitado.
> **Ações** são os processos que realizamos no RDD recém-criado, oriundos das transformações, para obter os resultados desejados. Exemplos de ações são *collect()* e *countByKey()*. A ação retorna o resultado para o driver.

Ainda em transformação, existem dois tipos que são **narrow** e **wide**:

> **Narrow transformation** não precisa que um **shuffle** seja executado entre as partições, ou seja, não precisa embaralhar os dados entre os nodes.
>
> **Wide transformation** exige que o shuffle ocorra.

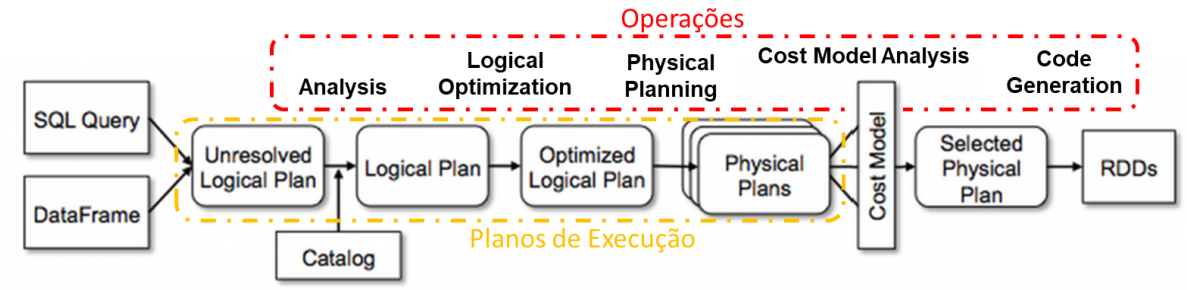
Logical e Physical Execution Plan

https://blog.knoldus.com/understanding-sparks-logical-and-physical-plan-in-laymans-term/

https://medium.com/datalex/sparks-logical-and-physical-plans-when-why-how-and-beyond-8cd1947b605a

Um plano de execução é o conjunto de operações executadas para traduzir uma instrução de linguagem de consulta (SQL, Spark SQL, operações Dataframe etc.) para um conjunto de operações lógicas e físicas otimizadas, ou seja, é um conjunto de operações que serão executadas da instrução SQL (ou Spark SQL) para o DAG que será enviado para Spark Executors.

No Spark, o otimizador é denominado *Catalyst* e pode ser representado pelo esquema abaixo e produzirá diferentes tipos de planos:



O objetivo de todas essas operações e planos é produzir automaticamente a maneira mais eficaz de processar sua consulta.

Como podemos visualizar os planos?

Para gerar planos, precisamos trabalhar com Dataframes, independentemente de serem gerados de um SQL ou dataframe raw. A função que usaremos, em Python, é a *explain()*. Por exemplo, se tivermos esses dois dataframes:

```
itemsSchema = ("id integer, name string, price float")
ordersSchema=("id integer, itemid integer, count integer")

items = spark.createDataFrame([[0, "Tomato", 2.0], \
                               [1, "Watermelon", 5.5], \
                               [2, "pineapple", 7.0]], \
                              schema=itemsSchema)

orders = spark.createDataFrame([[100, 0, 1], \
                                [100, 1, 1], \
                                [101, 2, 3], \
                                [102,2,8]],\
                               schema=ordersSchema)
```

Podemos manipulá-los da dessa forma:

```
from pyspark.sql.functions import sum

y=(items.join(orders,items.id==orders.itemid, how="inner"))\
        .where(items.id==2)\
        .groupBy("name","price").agg(sum("count")\
        .alias("c"))
```

Ou dessa:

```
items.createOrReplaceTempView("ITEMS")
orders.createOrReplaceTempView("ORDERS")

x=sql('''select ITEMS.name,
                ITEMS.price,
                SUM(ORDERS.count) as c
         from ITEMS, ORDERS
         where ITEMS.id=ORDERS.itemid
           and ITEMS.id=2
         group by ITEMS.name, ITEMS.price''')
```

Em ambas as formas podemos chamar o *explain():*

```
x.explain()
y.explain()
```

Por default, a saída será assim:

```
== Physical Plan ==
*(4) HashAggregate(keys=[name#765, price#766], functions=[finalmerge_sum(merge sum#817L) AS sum(cast(count#772 as bigint))#783L])
+- Exchange hashpartitioning(name#765, price#766, 200), true, [id=#4134]
   +- *(3) HashAggregate(keys=[name#765, knownfloatingpointnormalized(normalizenanandzero(price#766)) AS price#766], functions=[partial_sum(cast(count#772 as bigint)) AS sum#817L])
      +- *(3) Project [name#765, price#766, count#772]
         +- *(3) SortMergeJoin [id#764], [itemid#771], Inner
            :- Sort [id#764 ASC NULLS FIRST], false, 0
            :  +- Exchange hashpartitioning(id#764, 200), true, [id=#4123]
            :     +- *(1) Filter (isnotnull(id#764) AND (id#764 = 2))
            :        +- *(1) Scan ExistingRDD[id#764,name#765,price#766]
            +- Sort [itemid#771 ASC NULLS FIRST], false, 0
               +- Exchange hashpartitioning(itemid#771, 200), true, [id=#4127]
                  +- *(2) Project [itemid#771, count#772]
                     +- *(2) Filter ((itemid#771 = 2) AND isnotnull(itemid#771))
                        +- *(2) Scan ExistingRDD[id#770,itemid#771,count#772]
```

Há vários tipos de planos de execução. Antes do Spark 3.0 haviam apenas dois tipos:

explain(extended=False): mostra apenas o physical plan
explain(extended=True): mostra o logical e physical plan

Vejamos a saída com extended=True:

```
== Analyzed Logical Plan ==
name: string, price: float, c: bigint
Aggregate [name#2222, price#2223], [name#2222, price#2223, sum(cast(count#2229 as bigint)) AS c#2239L]
+- Filter ((id#2221 = itemid#2228) AND (id#2221 = 2))
   +- Join Inner
      :- SubqueryAlias items
      :  +- LogicalRDD [id#2221, name#2222, price#2223], false
      +- SubqueryAlias orders
         +- LogicalRDD [id#2227, itemid#2228, count#2229], false

== Optimized Logical Plan ==
Aggregate [name#2222, price#2223], [name#2222, price#2223, sum(cast(count#2229 as bigint)) AS c#2239L]
+- Project [name#2222, price#2223, count#2229]
   +- Join Inner, (id#2221 = itemid#2228)
      :- Filter (isnotnull(id#2221) AND (id#2221 = 2))
      :  +- LogicalRDD [id#2221, name#2222, price#2223], false
      +- Project [itemid#2228, count#2229]
         +- Filter ((itemid#2228 = 2) AND isnotnull(itemid#2228))
            +- LogicalRDD [id#2227, itemid#2228, count#2229], false

== Physical Plan ==
*(4) HashAggregate(keys=[name#2222, price#2223], functions=[finalmerge_sum(merge sum#2274L) AS sum(cast(count#2229 as bigint))#2240L], output=[name#2222, price#2223, c#2239L])
+- Exchange hashpartitioning(name#2222, price#2223, 200), true, [id=#5183]
   +- *(3) HashAggregate(keys=[name#2222, knownfloatingpointnormalized(normalizenanandzero(price#2223)) AS price#2223], functions=[partial_sum(cast(count#2229 as bigint)) AS sum#2274L], output=[name#2222, price#2223, sum#2274L])
      +- *(3) Project [name#2222, price#2223, count#2229]
         +- *(3) SortMergeJoin [id#2221], [itemid#2228], Inner
            :- Sort [id#2221 ASC NULLS FIRST], false, 0
            :  +- Exchange hashpartitioning(id#2221, 200), true, [id=#5172]
            :     +- *(1) Filter (isnotnull(id#2221) AND (id#2221 = 2))
            :        +- *(1) Scan ExistingRDD[id#2221,name#2222,price#2223]
            +- Sort [itemid#2228 ASC NULLS FIRST], false, 0
               +- Exchange hashpartitioning(itemid#2228, 200), true, [id=#5176]
                  +- *(2) Project [itemid#2228, count#2229]
                     +- *(2) Filter ((itemid#2228 = 2) AND isnotnull(itemid#2228))
                        +- *(2) Scan ExistingRDD[id#2227,itemid#2228,count#2229]
```

A partir do Spark 3.0 a função *explain()* ganhou um novo parâmetro que controla o resultado da saída e a formatação da mesma, o **mode**:

- **explain(mode=”simple”)** que mostra apenas o physical plan (semelhante ao extended=False).
- **explain(mode=”extended”)** mostra o logical e o physical plan (semelhante ao extended=True).
- **explain(mode=”codegen”)** irá exibir o código java planejado para ser executado.
- **explain(mode=”cost”)** exibirá o logical plan e os custos relativos a ele, caso exista.
- **explain(mode=”formatted”)** esse mostra apenas o physical plan mas de uma forma mais organizada.

Vejamos um exemplo:

```
x.explain(mode="formatted")
```

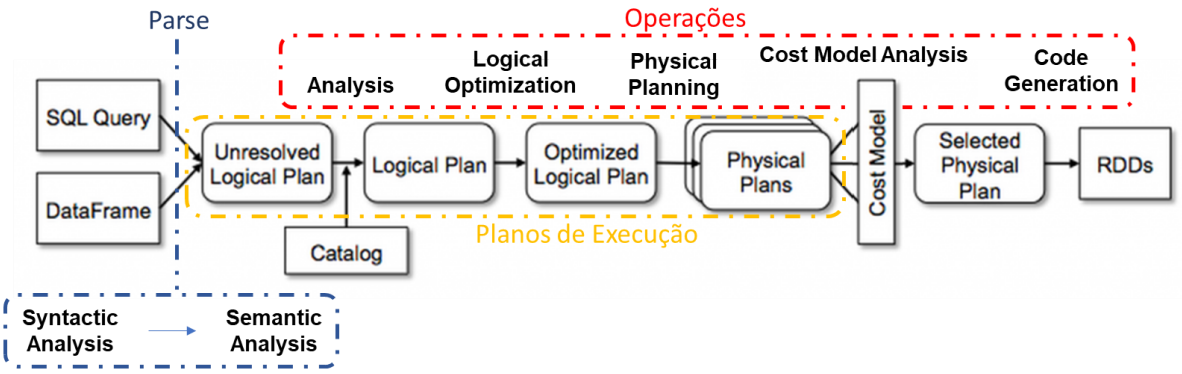
```
== Physical Plan ==
* HashAggregate (14)
+- Exchange (13)
   +- * HashAggregate (12)
      +- * Project (11)
         +- * SortMergeJoin Inner (10)
            :- Sort (4)
            :  +- Exchange (3)
            :     +- * Filter (2)
            :        +- * Scan ExistingRDD (1)
            +- Sort (9)
               +- Exchange (8)
                  +- * Project (7)
                     +- * Filter (6)
                        +- * Scan ExistingRDD (5)


(1) Scan ExistingRDD [codegen id : 1]
Output [3]: [id#2525, name#2526, price#2527]
Arguments: [id#2525, name#2526, price#2527], MapPartitionsRDD[602] at applySchemaToPythonRDD at <unknown>:0, ExistingRDD, UnknownPartitioning(0)

(2) Filter [codegen id : 1]
Input [3]: [id#2525, name#2526, price#2527]
Condition : (isnotnull(id#2525) AND (id#2525 = 2))

(3) Exchange
Input [3]: [id#2525, name#2526, price#2527]
Arguments: hashpartitioning(id#2525, 200), true, [id=#6140]
```

Vamos voltar ao otimizador denominado **Catalyst**, como dito anteriormente e é representado pelo fluxo abaixo:



Precisamos entender todos os planos de execução envolvidos nesse processo e, para isso, iremos partir do Dataframe. Quando envolvemos os dataframes, eles passam por um processo chamado **Parse** que faz algumas verificações como **Análise Sintática** e **Análise Semântica**, veja a diferença:

- **Análise sintática** verifica se o código foi escrito corretamente e se não há erros de sintaxe nele, ou seja, analisa o código em busca de erros de programação, logo depois irá executar a
- **Análise semântica** que verifica se os objetos mencionados no código realmente existem e se o usuário que irá submeter o job tem permissão sobre eles.

Como resultado do Parse, teremos o **Unresolved Logical Plan** que apresentará a seguinte estrutura:

```
== Parsed Logical Plan ==
'Aggregate ['name, 'price], [unresolvedalias('name, None), unresolvedalias('price, None), sum('count) AS c#2709]
+- Filter (id#2661 = 2)
   +- Join Inner, (id#2661 = itemid#2668)
      :- LogicalRDD [id#2661, name#2662, price#2663], false
      +- LogicalRDD [id#2667, itemid#2668, count#2669], false
```

Analisando esse plano de execução podemos entender o porquê do nome (**unresolved**). Isso porque nem todos os objetos, colunas, tipos etc foram encontrados (resolvidos). Observe que as colunas *name* e *price* bem como seus respectivos tipos, incluindo o tipo da coluna *count*, ainda são desconhecidos nesse plano enquanto que outras colunas já são conhecidas (resolvidas).

Após esse plano de execução ser gerado, o próximo passo é executar a operação **Analysis** e gerar o **Logical Plan** ou **Analyzed Logical Plan**. Essa operação acessa uma estrutura interna do Spark chamada **Catalog** (catálogo) que pode ser entendido como o metastore ou dicionário de dados que ele armazena dos schemas dos dataframes anteriores.

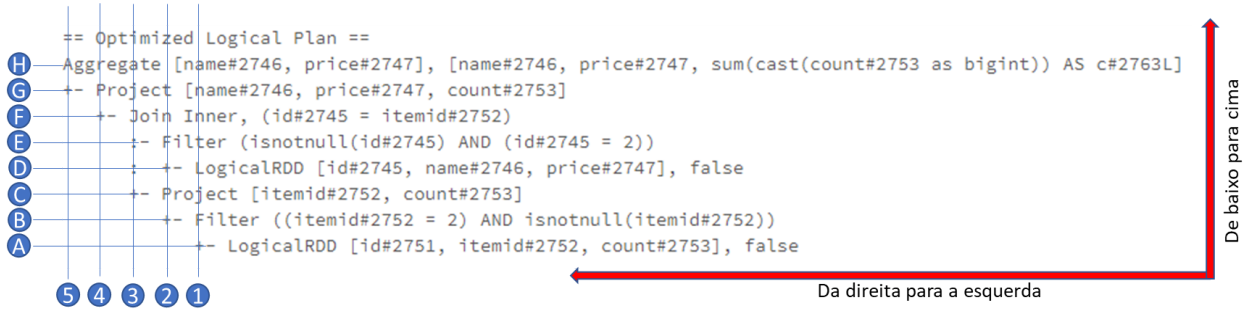
```
== Analyzed Logical Plan ==
name: string, price: float, c: bigint
Aggregate [name#2662, price#2663], [name#2662, price#2663, sum(cast(count#2669 as bigint)) AS c#2709L]
+- Filter (id#2661 = 2)
   +- Join Inner, (id#2661 = itemid#2668)
      :- LogicalRDD [id#2661, name#2662, price#2663], false
      +- LogicalRDD [id#2667, itemid#2668, count#2669], false
```

Agora que o plano Lógico foi gerado, ele será otimizado com base em várias regras aplicadas em operações de filtros, agregação etc. Essas operações serão reordenadas para otimizar o plano lógico. Quando a otimização

terminar, ela produzirá esta saída que é denominada **Optimized Logical Plan**:

```
== Optimized Logical Plan ==
Aggregate [name#2746, price#2747], [name#2746, price#2747, sum(cast(count#2753 as bigint)) AS c#2763L]
+- Project [name#2746, price#2747, count#2753]
   +- Join Inner, (id#2745 = itemid#2752)
      :- Filter (isnotnull(id#2745) AND (id#2745 = 2))
      :  +- LogicalRDD [id#2745, name#2746, price#2747], false
      +- Project [itemid#2752, count#2753]
         +- Filter ((itemid#2752 = 2) AND isnotnull(itemid#2752))
            +- LogicalRDD [id#2751, itemid#2752, count#2753], false
```

Acima está o novo plano lógico de execução otimizado, mas precisamos entender melhor como lê-lo e interpretá-lo para que possamos compreender o que foi feito. Para isso, vamos voltar ao código escrito em python e ao mais recente optmized logical plan, mas antes de tudo, temos que entender como ler o plano lógico.

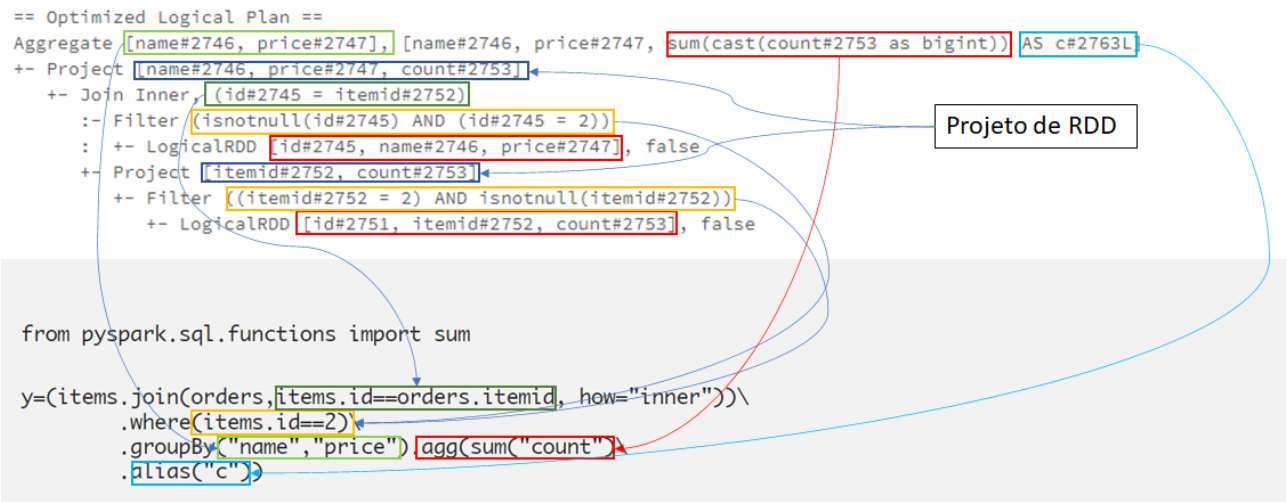


A leitura é feita seguindo algumas regras pré-estabelecidas:

- da direita para esquerda
- de baixo para cima
- obedecendo a sequência das operações

Vejamos, seguindo as regras supracitadas, a primeira operação a ser executada é a leitura do RDD *orders* contido em **A1**. Logo em seguida, duas outras operações simultâneas são executadas: **B2** que é o filtro do *itemId* do dataset *orders* e **D2** que é a leitura do outro RDD, o *items*. Depois disso, temos o **C3** que representa um dataset oriundo das operações que ocorreram e o **E3** que também é o filtro do *itemId* mas dessa vez do dataset *items*. O **F4** é o *join* dos dois datasets e **G5** representa um dataset resultado do *join*. Por fim, o **H5** contém a função de agregação.

Vamos analisar novamente mas agora com outra perspectiva:



Esse emaranhado de ligações pode parecer, a princípio, muito confuso, mas vamos analisá-lo com calma para compreender melhor.

Antes de iniciarmos, observe que o otimizador "quebrou" o predicado “where” em duas partes, ou seja, apesar de no código escrito em python ter apenas uma cláusula where, o Catalyst Optimizer “achou por bem” que o plano mais otimizado seria esse. Mais a frente falaremos sobre os planos de execuções que o Catalyst gera.

Podemos ver no Optimized Logical Plan que ele inicia trazendo, do dataset *orders*, apenas as colunas (*id, itemid* e *count*) que serão utilizadas em todos os processos do plano. Logo em seguida ele filtra os dados pelo itemid e ainda garante que nenhum deles seja nulo, desse mesmo dataset. Esse filtro antecipado é interessante pois faz com que a volumetria dos dados seja menor nas fases subsequentes. Depois, ao mesmo tempo em que ele gera um projeto de RDD com o resultado desse filtro, também é selecionada apenas as colunas utilizadas no dataset *items* (*id, name* e *price*). Adiante, os dados também são filtrados pelo id do dataset items e logo em seguida, gerado um outro projeto de RDD com esse resultado.

Com os dois RDDs gerados como resultado dos filtros, é feito o join entre eles e logo em seguida a operação de agregação.

Neste ponto temos o Optmized Logical Plan gerado e já sabemos interpretá-lo corretamente. Agora, o **Catalyst Optmizer** irá gerar vários Physical Execution Plans com estimativas de tempo de execução utilizaçao de recursos, mas somente um será escolhido. Abaixo está um exemplo de um Physical Plan:

```
== Physical Plan ==
*(4) HashAggregate(keys=[name#10, price#11], functions=[finalmerge_sum(merge sum#72L) AS sum(cast(count#17
as bigint))#50L], output=[name#10, price#11, c#51L])
+- Exchange hashpartitioning(name#10, price#11, 200), true, [id=#299]
   +- *(3) HashAggregate(keys=[name#10, knownfloatingpointnormalized(normalizenanandzero(price#11)) AS pric
e#11], functions=[partial_sum(cast(count#17 as bigint)) AS sum#72L], output=[name#10, price#11, sum#72L])
      +- *(3) Project [name#10, price#11, count#17]
         +- *(3) SortMergeJoin [id#9], [itemid#16], Inner
            :- Sort [id#9 ASC NULLS FIRST], false, 0
            :  +- Exchange hashpartitioning(id#9, 200), true, [id=#288]
            :     +- *(1) Filter (isnotnull(id#9) AND (id#9 = 2))
            :        +- *(1) Scan ExistingRDD[id#9,name#10,price#11]
            +- Sort [itemid#16 ASC NULLS FIRST], false, 0
               +- Exchange hashpartitioning(itemid#16, 200), true, [id=#292]
                  +- *(2) Project [itemid#16, count#17]
                     +- *(2) Filter ((itemid#16 = 2) AND isnotnull(itemid#16))
                        +- *(2) Scan ExistingRDD[id#15,itemid#16,count#17]
```

As regras de leitura são as mesmas do Optimized Logical Plan.

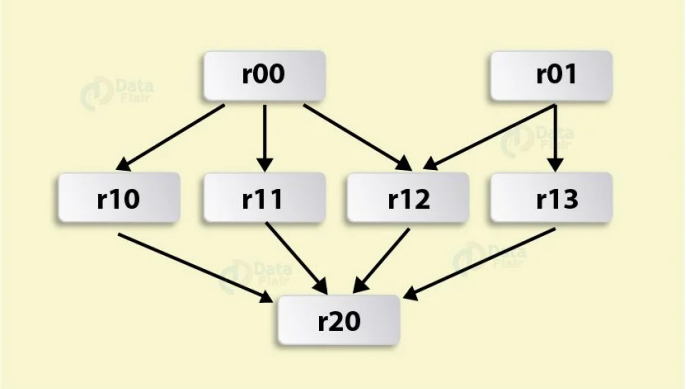
Vale lembrar que a partir do Spark 3.0 foi implementado o **AQE** (*Adaptative Query Execution*) que nada mais é do que um plano de execução alternativo, gerado em tempo de execução, ou seja, quando o job já está em execução, e o Spark determina se esse é o melhor plano, naquele momento. Sendo assim, ele muda a estratégia em tempo de execução.

RDD Lineage (linhagem)

Como vimos, um RDD é lazy por natureza. Isso significa que uma série de transformações é executada em um RDD mas não imediatamente.
Enquanto criamos um novo RDD a partir de um Spark RDD existente, esse novo RDD também carrega um ponteiro para o RDD de origem. Isso ocorre para todas as dependências entre os RDDs que são registrados em um gráfico. Esse gráfico é chamado de **RDD Lineage**.
O RDD Lineage nada mais é do que o gráfico de todos os RDDs pai de um RDD. Também o chamamos de gráfico de **RDD operator** ou **gráfico de**

dependência RDD. Para ser mais específico, o RDD Lineage é o resultado em modo gráfico de transformações aplicadas no Spark.

Abaixo temos um exemplo de um RDD Lineage:



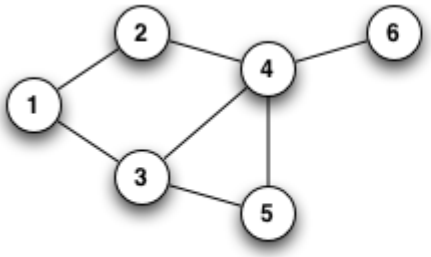
DAG (Directed Acyclic Graph)

O DAG (Directed Acyclic Graph) e o Physical Execution Plan são os principais conceitos do Apache Spark. Entender isso pode nos ajudar a codificar aplicativos Spark mais performáticos.

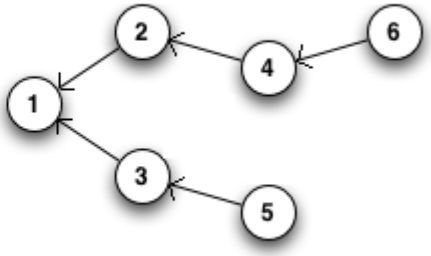
O que é um DAG de acordo com a Teoria dos Gráficos?

Em matemática, um Grafo Acíclico Dirigido (Directed Acyclic Graph) ou DAG, é um gráfico sem ciclo, ou seja, para qualquer vértice, não há nenhuma ligação começando e acabando nele além de ser direcionado, orientado ou dirigido.

Antes de tudo vamos entender a diferença entre dirigido e não dirigido. Vejamos os exemplos abaixo:



Não Dirigido

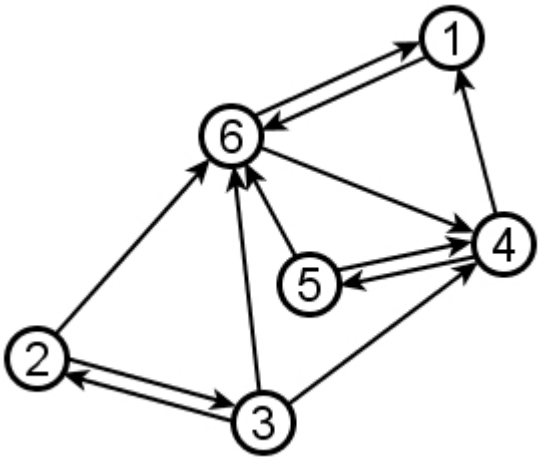


Dirigido

Observe que o gráfico em si já é autoexplicativo. Os gráficos não dirigidos não possuem setas, ou seja, seus vértices podem apontar para qualquer direção enquanto que, no gráfico dirigido, há uma direção específica a ser tomada, é direcionado, orientado.

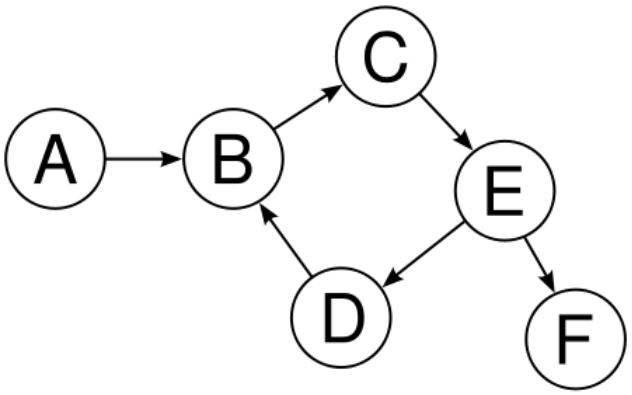
Agora que já entendemos essa diferença, vamos compreender outro conceito. Na matemática existem gráficos cíclicos e acíclicos.

Os gráficos cíclicos são mais complexos e difíceis de interpretar enquanto que o acíclico são mais simples e fáceis de entender. Vamos ver um exemplo de gráficos cíclicos:



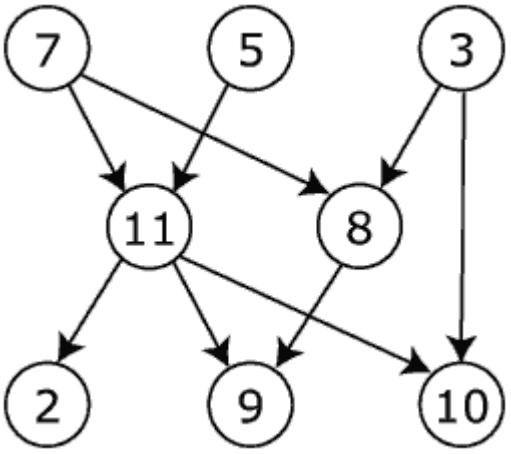
Observe que, partindo de qualquer vértice, independente da direção, é possível retornar ao vértice de origem, por exemplo, partindo do vértice 1 chegamos ao 6. Daqui, a direção pode tomar dois caminhos: para o vértice 4

ou voltar para o vértice 1. Se formos para o vértice 4 ainda poderemos retornar para o vértice 1. Vejamos outro exemplo:



Partindo do vértice A andamos em direção ao vértice B, depois ao C, ao E e, agora, se partirmos para o vértice D veremos que ele retorna ao B. Isso é uma característica de um gráfico cíclico. Dependendo da quantidade de vértices e arestas ele pode tornar sua compreensão difícil.

Agora vejamos um exemplo de um gráfico acíclico:



Muito mais fácil de entender, não? Aqui podemos partir de qualquer um dos três vértices: 7, 5 ou 3 e claramente observamos que é um caminho sem volta ou seja, em momento algum é possível retornar ao vértice de origem.

Onde um DAG atua no Apache Spark?

Ao contrário do Hadoop, onde o usuário precisa dividir o job inteiro em pequenos jobs e amarrá-los para encontrar o MapReduce, o Spark Driver identifica as tarefas implicitamente que podem ser calculadas em paralelo com os dados particionados no cluster. Com essas tarefas identificadas, o Spark Driver constrói um fluxo lógico de operações que podem ser representadas em um DAG. Assim, o Spark constrói seu próprio plano de execução implicitamente a partir da aplicação submetida.

collect_list() e collect_set()

Essas funções são utilizadas, geralmente, em agregações, seja com group by ou utilizando o window partition. Elas agregam uma coluna trazendo todos os seus valores como um array. Imagine um dataframe como mostrado abaixo:

```
+----------+--------------+
|name      |booksInterested|
+----------+--------------+
|James     |Java          |
|James     |C#            |
|James     |Python        |
|Michael   |Java          |
|Michael   |PHP           |
|Michael   |PHP           |
|Robert    |Java          |
|Robert    |Java          |
|Robert    |Java          |
|Washington|null          |
+----------+--------------+
```

Agora veremos a utilização do collect_list:

```
val df2 = df.groupBy("name").agg(collect_list("booksIntersted")
  .as("booksInterested"))
df2.printSchema()
df2.show(false)

root
 |-- name: string (nullable = true)
 |-- booksInterested: array (nullable = true)
 |    |-- element: string (containsNull = true)

+----------+------------------+
|name      |booksInterested   |
+----------+------------------+
|James     |[Java, C#, Python]|
|Washington|[]                |
|Michael   |[Java, PHP, PHP]  |
|Robert    |[Java, Java, Java]|
+----------+------------------+
```

Observe que a função agregou, juntou, reuniu todos os valores da coluna **booksInterested** em um array.

Agora vejamos a utilização do collect_set:

```
df.groupBy("name").agg(collect_set("booksInterested")
  .as("booksInterestd"))
  .show(false)

+----------+------------------+
|name      |booksInterested   |
+----------+------------------+
|James     |[Java, C#, Python]|
|Washington|[]                |
|Michael   |[PHP, Java]       |
|Robert    |[Java]            |
+----------+------------------+
```

Observe que a diferença é que o *collect_set* deduplicou os valores da coluna booksInterested, ou seja, eliminou os valores duplicados.

Essa é a única diferença!

cache() e persist()

Abaixo estão as vantagens de usar os métodos Spark *cache()* e *persist().*

Custo eficiente – o trabalho de computação do Spark é muito caro, portanto, a reutilização desse trabalho é usada para economizar custos.

Tempo eficiente – Reutilizar processamento que precisam ser repetidos economiza muito tempo.

Tempo de execução – Economiza tempo de execução do job e podemos executar mais jobs no mesmo cluster.

Quando você persiste um conjunto de dados, cada nó armazena seus dados particionados na memória e os reutiliza em outras ações nesse conjunto de dados. E os dados persistentes do Spark nos nós são tolerantes a falhas, o que significa que se alguma partição de um Dataset for perdida, ela será recalculada automaticamente usando as transformações originais que a criaram.

Abaixo, explicarei como usar o *cache()* e *persist()* com **DataFrame** ou **Dataset**.

Spark Cache

O cache do DataFrame ou Dataset, por padrão, é com storage level **MEMORY_AND_DISK.** Por outro lado, o storage level padrão para o RDD é **MEMORY_ONLY**.

eder.sferreira@gmail.com

Syntaxe

```
cache() : Dataset.this.type
```

O método *cache()* na classe *Dataset* chama internamente o método *persist()* que, por sua vez, usa *sparkSession.sharedState.cacheManager.cacheQuery* para armazenar em cache o conjunto de resultados de DataFrame ou Dataset. Vejamos um exemplo.

Exemplo

```
val spark:SparkSession = SparkSession.builder()
   .master("local[1]")
   .appName("SparkByExamples.com")
   .getOrCreate()
 import spark.implicits._
 val columns = Seq("Seqno","Quote")
 val data = Seq(("1", "Be the change that you wish to see in the world"),
   ("2", "Everyone thinks of changing the world, but no one thinks of changing himself."),
   ("3", "The purpose of our lives is to be happy."))
 val df = data.toDF(columns:_*)

 val dfCache = df.cache()
 dfCache.show(false)
```

Spark Persist

O *persist()* tem duas assinaturas, a primeira assinatura não aceita nenhum argumento que, por padrão, a salva no *storage level* **MEMORY_AND_DISK** e a segunda assinatura, usa *StorageLevel* como argumento para definir um dos diferentes níveis de armazenamento.

Syntax

```
1) persist() : Dataset.this.type
2) persist(newLevel : org.apache.spark.storage.StorageLevel) : Dataset.this.type
```

Exemplo

```
val dfPersist = df.persist()
dfPersist.show(false)
```

Usando a segunda assinatura, você pode salvar DataFrame/Dataset em um dos storage levels **MEMORY_ONLY**, **MEMORY_AND_DISK**, **MEMORY_ONLY_SER**, **MEMORY_AND_DISK_SER**, **DISK_ONLY**, **MEMORY_ONLY_2**,**MEMORY_AND_DISK_2**

```
val dfPersist = df.persist(StorageLevel.MEMORY_ONLY)
dfPersist.show(false)
```

De acordo com o exemplo acima, ele armazena DataFrame/Dataset na memória

Spark Unpersist

Também podemos excluir a persistência de um DataFrame ou Dataset.

Sintaxe

```
unpersist() : Dataset.this.type
unpersist(blocking : scala.Boolean) : Dataset.this.type
```

Exemplo

```
val dfPersist = dfPersist.unpersist()
dfPersist.show(false)
```

O método unpersist(Boolean = false) com um valor booleano como argumento ele pode tornar o método síncrono, ou seja, bloqueia o processo até que todos os blocos sejam deletados

Storage Level de persist()

Todos os diferentes storage levels suportados do Spark estão disponíveis na classe org.apache.spark.storage.StorageLevel. O storage level especifica como e onde persistir ou armazenar em cache um DataFrame ou Dataset.

MEMORY_ONLY – Este é o comportamento padrão do método RDD cache() e armazena o RDD ou DataFrame como objetos **desserializados** na memória JVM. Quando não houver memória suficiente disponível, ele não salvará o DataFrame de algumas partições e estas serão recalculadas quando necessário. Isso consome mais memória, mas ao contrário do RDD, isso seria mais lento do que o *storage level* **MEMORY_AND_DISK**, pois reprocessa as partições não salvas e esse re-processamento é dispendioso.

MEMORY_ONLY_SER – É o mesmo que **MEMORY_ONLY**, mas a diferença é que ele armazena RDD como objetos **serializados** na memória JVM. Leva menos memória (com eficiência de espaço) do que **MEMORY_ONLY**, pois salva objetos como serializados mas leva mais alguns ciclos de CPU adicionais para desserializar.

MEMORY_ONLY_2 – Igual ao *storage level* **MEMORY_ONLY**, mas replica cada partição para dois nós de cluster.

MEMORY_ONLY_SER_2 – Igual ao *storage level* **MEMORY_ONLY_SER**, mas replica cada partição para dois nós de cluster.

MEMORY_AND_DISK – Este é o comportamento padrão do DataFrame ou Dataset. Neste *storage level*, o DataFrame será armazenado na memória JVM como um objeto **desserializado**. Quando o armazenamento necessário é maior que a memória disponível, ele armazena algumas das partições em excesso no disco e lê os dados do disco quando necessário. É mais lento, pois há E/S envolvida.

MEMORY_AND_DISK_SER – É o mesmo que o *storage level* **MEMORY_AND_DISK**, a diferença é que **serializa** os objetos DataFrame na memória e no disco quando não há espaço disponível.

eder.sferreira@gmail.com

MEMORY_AND_DISK_2 – Igual ao *storage level* **MEMORY_AND_DISK**, mas replica cada partição para dois nós de cluster.

MEMORY_AND_DISK_SER_2 – Igual ao *storage level* **MEMORY_AND_DISK_SER**, mas replica cada partição para dois nós de cluster.

DISK_ONLY – Neste *storage level*, o DataFrame é armazenado apenas em disco e o tempo de processamento da CPU é alto, pois a E/S está envolvida.

DISK_ONLY_2 – Igual ao *storage level* **DISK_ONLY**, mas replica cada partição para dois nós de cluster.

Abaixo está a tabela de representação do *storage level* fazendo a comparação entre o espaço utilizado, tempo de cpu, local de armazenamento, tipo de armazenamento (serializado ou desserializado) e se há possibilidade de reprocessamento.

Storage Level	Space used	CPU time	In memory	On-disk	Serialized	Recompute some partitions
MEMORY_ONLY	High	Low	Y	N	N	Y
MEMORY_ONLY_SER	Low	High	Y	N	Y	Y
MEMORY_AND_DISK	High	Medium	Some	Some	Some	N
MEMORY_AND_DISK_SER	Low	High	Some	Some	Y	N
DISK_ONLY	Low	High	N	Y	Y	N

Alguns pontos a serem observados em persist

- O Spark monitora automaticamente todas as chamadas *persist()* e *cache()* que é feito e verifica o uso em cada nó e descarta os dados persistentes se não forem usados ou usando o algoritmo LRU (Last Recently Used). Conforme discutido em uma das seções acima, você também pode remover manualmente usando o método *unpersist().*
- O *persist()* e *cache()* são apenas duas das técnicas de otimização para melhorar o desempenho dos trabalhos do Spark.
- Para RDD cache() o *storage level* padrão é **MEMORY_ONLY** mas, para DataFrame e Dataset, o padrão é **MEMORY_AND_DISK.**

- No Spark UI, a guia Armazenamento mostra onde existem partições na memória ou no disco no cluster.
- Dataset cache() é um alias para persist(StorageLevel.MEMORY_AND_DISK)
- O armazenamento em cache do Spark DataFrame ou Dataset é uma operação **lazy**, o que significa que um DataFrame não será armazenado em cache até que uma ação seja acionada.

Spark Tuning

Antes de iniciarmos os estudos de tuning, precisamos entender a diferença entre RDD, Dataframe e Dataset. Qual desses oferece melhor desempenho e funcionalidades?

RDD

- Processamento em memória: O PySpark carrega os dados do disco e processa-os na memória e mantém os dados na memória também, essa é a principal diferença entre o PySpark e o Mapreduce (alto E/S). Entre as transformações, também podemos armazenar em *cache/persist* o RDD para reutilizar os processamentos anteriores.
- É imutável.
- É tolerante à falhas, ou seja, no caso de uma falha, ele se recupera reprocessando os dados de outra partição.
- Trabalha no modo lazy.
- Por default, um RDD é criado com particionamento e o número de partições é a quantidade de cores disponíveis ou conforme a configuração do contexto Spark.

Há duas formas de criar um RDD:

- Paralelizando um collection existente
- Referenciando um dataset de um armazenamento externo (HDFS, S3, etc)

Antes de continuarmos, primeiro vamos inicializar o *SparkSession* usando o método padrão da classe SparkSession. Ao inicializar, precisamos fornecer o nome do *master* e o *appName* conforme mostrado abaixo.

```
from pyspark.sql import SparkSession
spark:SparkSession = SparkSession.builder()
      .master("local[1]")
      .appName("SparkByExamples.com")
      .getOrCreate()
```

eder.sferreira@gmail.com

master() – Caso esteja rodando em um cluster, é necessário utilizar o nome do *master* como argumento em *master()*. No entanto, se estiver executando em modo standalone, utilize o local[x] onde o x deve ser um número inteiro maior que 0. e representa a quantidade de partições que os RDDs, Dataframes e Datasets terão, nesta ssessão. O O valor ideal para x é a quantidade de cores de CPU existente.

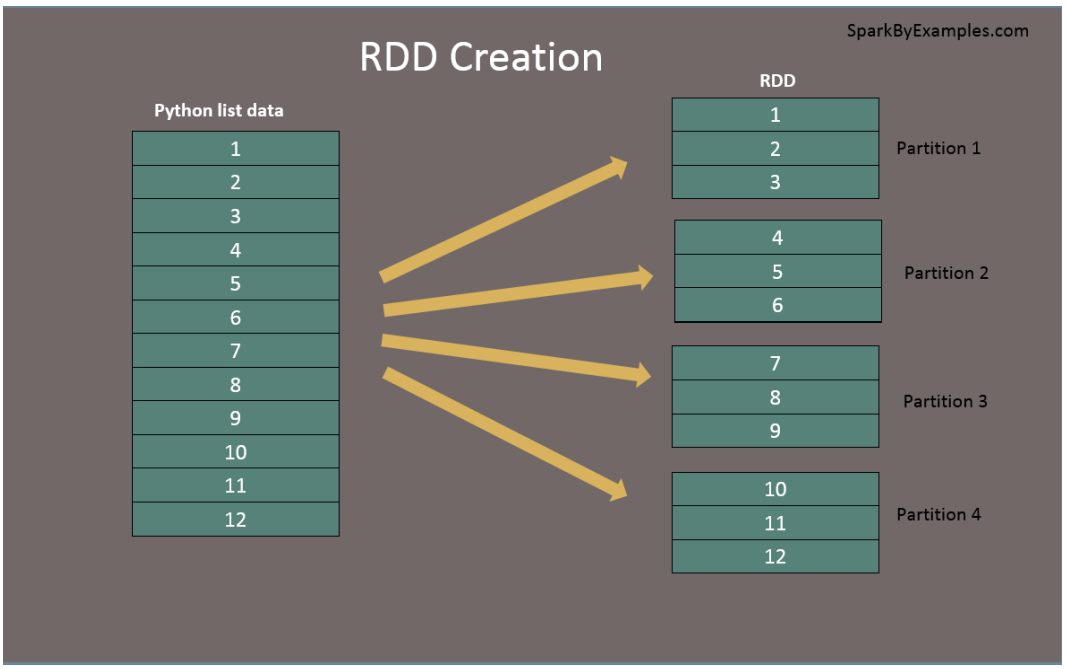
appName() – Define o nome da aplicação.

getOrCreate() – Esta função retorna um objeto do tipo SparkSession. Caso o objeto já exista, ele retorna, caso contrário ele cria.

SparkContext

Paralelizando um collection existente

Usando a função parallelize() de SparkContext (sparkContext.parallelize()), você pode criar um RDD. Esta função carrega a coleção existente de seu programa de driver para paralelizar o RDD. Este é um método básico para criar RDD mas só pode ser usado quando você já possui dados na memória carregados de um arquivo ou de um banco de dados. e exigia que todos os dados estivessem presentes no programa de driver antes de criar o RDD.




```
#Create RDD from parallelize
data = [1,2,3,4,5,6,7,8,9,10,11,12]
rdd=spark.sparkContext.parallelize(data)
```

Para aplicativos de produção, criamos principalmente RDD usando sistemas de armazenamento externo como HDFS, S3, HBase e etc. Para simplificar este exemplo do PySpark RDD, estamos usando arquivos do sistema local ou carregando-os da lista python.

Referenciando um dataset de um armazenamento externo

asdf

Dataframe

asdf

Dataset

eder.sferreira@gmail.com

asdf

https://sparkbyexamples.com/spark/spark-performance-tuning/

O ajuste e a otimização do desempenho do Spark é um tópico maior que consiste em várias técnicas e configurações (memória e núcleos de recursos), aqui eu abordei algumas das melhores diretrizes que usei para melhorar minhas cargas de trabalho e continuarei atualizando isso à medida que deparar-se com novos caminhos.

- Use DataFrame/Dataset ao invés de RDD
- Use coalesce() ao invés de repartition()
- Use mapPartitions() ao invés de map()
- Use formatos de dados serializados
- Evite UDFs (Funções Definidas pelo Usuário)
- Cache de dados na memória
- Reduza as dispendiosas operações Shuffle
- Desativar gravações de DEBUG e INFO

Use Dataframe/Dataset ao invés de RDD

Para trabalhos do Spark, prefira usar Dataset/DataFrame em vez de RDD, pois o Dataset e o DataFrame incluem vários módulos de otimização para melhorar o desempenho das cargas de trabalho do Spark.

eder.sferreira@gmail.com

O RDD é uma parte da construção da programação do Spark, mesmo quando usamos DataFrame/Dataset, o Spark usa internamente o RDD para executar operações/consultas mas de maneira eficiente e otimizada, analisando sua consulta e criando o plano de execução graças ao otimizador do projeto **Tungsten** e **Catalyst**.

Por que o RDD é lento?

O uso do RDD leva diretamente a problemas de desempenho, pois o Spark não sabe como aplicar as técnicas de otimização no RDD (apenas no Dataframe/Dataset) que serializa e desserializa os dados quando distribui em um cluster (repartition e shuffling).

Serialização e desserialização são operações **muito caras** para aplicativos Spark ou qualquer sistema distribuído, a maior parte do tempo é gasto apenas na serialização de dados, em vez de executar as operações, portanto, tente evitar o uso de RDD.

Como o DataFrame mantém a estrutura dos dados e tipos de coluna (como uma tabela RDBMS), ele pode lidar melhor com os dados armazenando e gerenciando com mais eficiência.

Use coalesce() ao invés de repartition()

O *coalesce()* é uma versão otimizada ou aprimorada de *repartition()*.

Caso queira **reduzir** o número de partições, use o *coalesce()*.

Caso queira **aumentar** o número de partições, use *repartition().*

Exemplo de repartição()

eder.sferreira@gmail.com