

## Parte 6.2: Joins, Shuffles, Broadcasts, Accumulators

# Revisitando o “join”

- Considere os seguintes Datasets

```
> val largeDF = // Considere um DF com 1 milhão de linhas
> largeDF.repartition(100)
> largeDF.limit(3).show
```

```
+---+---+
|num|bit|
+---+---+
|  1|  1|
|  2|  0|
|  3|  1|
```

```
> val smallDF = // Considere um DF com 2 linhas
smallDF.show
> smallDF.show
```

```
+---+-----+
|bit|bitName|
+---+-----+
|  0|   zero|
|  1|   one|
+---+-----+
```

# Considere um “join” padrão

- Iremos fazer o “join” entre os nossos dois DataFrames — isso é um problema
  - O processo sort-merge-join faz o shuffle de todas as linhas do nosso largeDF
    - Pela chave de “bits” (a coluna de “join”) – Muito processamento para o shuffle
  - Chaves insuficientes para paralelismo !
    - Termina com apenas 2 partições não vazias
    - Adicionar mais nós workers não faz NENHUMA DIFERENÇA para ajudar neste trabalho

```
> largeDF.join(smallDF, largeDF("bit") === smallDF("bit")).explain
== Physical Plan ==
*SortMergeJoin [bit#1345], [bit#1338], Inner
:- *Sort [bit#1345 ASC NULLS FIRST], false, 0
:  +- Exchange hashpartitioning(bit#1345, 200)
:     +- *Filter isNotNull(bit#1345)
:        +- Scan ExistingRDD[num#1344,bit#1345]
+- *Sort [bit#1338 ASC NULLS FIRST], false, 0
:  +- Exchange hashpartitioning(bit#1338, 200)
:     +- *Filter isNotNull(bit#1338)
:        +- *SerializeFromObject [...]
:           +- Scan ExternalRDDScan[obj#1337]
```

# Reduzindo o Shuffle: Compartilhar dados do App

- Uma técnica é remover o shuffle completamente
  - Enviando dados do client (driver)
  - Veja a seguir

```
> val largeDF = // 0 mesmo do slide anterior
> case class Bit(bit:Int, bitName: String)
> val smallArray = List (Bit(0,"zero"), Bit(1,"one"))
> case class Result(num:Int, bit:Int, bitName:String)
> val mappedDF = largeDF.map(r => {
  if ( r.getInt(1) == smallArray(0).bit) {
    Result(r.getInt(0), r.getInt(1), smallArray(0).bitName)
  } else {
    Result(r.getInt(1), r.getInt(1), smallArray(1).bitName)
  } } )
> mappedDF.limit(2).show
```

num	bit	bitName
1	1	one
2	0	zero

# Plano Físico de Dados Compartilhados

- O plano abaixo mostra que não existe um processo de shuffle
  - Mas a cada vez que o `smallArray` é usado, ele é **enviado através da rede**
  - Para dados grandes (e.g. 20MB) usados muitas vezes, isso pode se tornar um ponto de sobrecarga
- Também há muita atividade de serialização
  - Um objeto Java é criado para o nosso lambda (no formato Tungsten)
  - Então, ele é serializado novamente para o formato Tungsten

```
> mappedDF.explain
== Physical Plan ==
*SerializeFromObject [...]
+- *MapElements <function1>, obj#21: $line24.$read$$iw$$iw$Result
+- *DeserializeToObject createexternalrow(...)
+- Scan ExistingRDD[num#2,bit#3]
```

# Variáveis de Transmissão para Compartilhamento de Dados

- **Variáveis de Transmissão** incluem dados apenas para leitura no cache de cada nó
  - E pode ser reutilizado sem nenhum custo
  - Cria a partir do método **SparkContext.broadcast()**
  - Acesse a partir da chamada **variable.value**
  - Variáveis de Transmissão são **imutáveis**
    - Mudanças NÃO são propagadas
  - Demonstraremos o uso de variável de transmissão para o `smallArray`

```
// largeDF smallArray, e a classe Result declaradas anteriormente
// Criando uma variável de Transmissão
> val smallArrayBC = sc.broadcast(smallArray)
// Usando-a no Map
> val mappedDF = largeDF.map(r => {
  if ( r.getInt(1) == smallArrayBC.value(0).bit) {
    Result(r.getInt(0), r.getInt(1), smallArrayBC.value(0).bitName)
  } else {
    Result(r.getInt(1), r.getInt(1), smallArrayBC.value(1).bitName)
  } } )
```

# Catalyst e Variáveis de Transmissão

- O Catalyst pode fazer a transmissão da variável para você
  - Você pode passar uma instrução de transmissão
    - `broadcast()` é a função que faz isso
- O Catalyst utiliza o objeto **BroadcastHashJoin**
  - Faz a transmissão do `smallArrayDF`
  - **Sem serialização**
  - Este é de longe o plano mais eficiente

```
// largeDF e smallDF declarados anteriormente
import org.apache.spark.sql.functions.broadcast
> largeDF.join(broadcast(smallDF),
               largeDF("bit") === smallDF("bit")).explain
== Physical Plan ==
*BroadcastHashJoin [bit#3], [bit#60], Inner, BuildRight
:- *Filter isNotNull(bit#3)
:  +- Scan ExistingRDD[num#2,bit#3]
+- BroadcastExchange HashedRelationBroadcastMode(...))
   +- LocalTableScan [bit#60, bitName#61]
```

# A Transmissão Automática do Catalyst

- O Catalyst pode automaticamente escolher uma transmissão
  - Caso o tamanho do seu dataframe seja menor que o parâmetro configurado **spark.sql.autoBroadcastJoinThreshold** (por padrão é 10MB)
  - O Catalyst pode dizer o tamanho de seus dados
  - Anteriormente, desativamos isso definindo o limite para -1

```
// largeDF e smallDF declarados anteriormente

// Definindo explicitamente o limite para 10MB
> spark.conf.set("spark.sql.autoBroadcastJoinThreshold", 1024*1024*10)

> largeDF.join(smallDF, largeDF("bit") === smallDF("bit")).explain
== Physical Plan ==
*BroadcastHashJoin [bit#3], [bit#8], Inner, BuildRight
:- *Filter isnotnull(bit#3)
: +- Scan ExistingRDD[num#2,bit#3]
+- BroadcastExchange HashedRelationBroadcastMode(...)
   +- LocalTableScan [bit#8]
```



# Acumuladores para Cálculo Compartilhado

- **Acumuladores** são variáveis que podem ser adicionadas em paralelo
  - Adicionado por meio de operadores associativos especiais(+=, add)
  - Os workers podem adicionar à variável, apenas o client pode lê-la
  - Pode ser usada para contadores ou somas
  - Abaixo temos um exemplo

```
> val accum = sc.accumulator(0, "My Accumulator")
accum: org.apache.spark.Accumulator[Int] = 0

> sc.parallelize(Array(1, 2, 3, 4)).
    foreach(x => accum += x)

> accum.value
res5: Int = 10
```

# Resumo de Transferências

- São úteis para reduzir os processos de shuffle
  - Mas lembre-se — dados de Transferências devem caber na memória
  - E se não estiver sendo usado mais de uma vez, pode ser bom passá-lo na transformação ao invés de fazer a Transferência
- O Catalyst pode te ajudar!
  - Tem otimizações que serão transmitidas automaticamente quando apropriado
  - Ou você pode fazer isso de forma manual

## Lab 6.2: Join e Broadcasts

## Parte 6.4: Orientações Gerais

# Use a UI do Spark

- Ele irá apontar os problemas
  - Vimos isso em muitos labs e slides (porta 4040)
- Coisas para olhar
  - Tempo de execução de uma Task
  - Quantidade de dados em um shuffle
  - Partições e informações sobre os volumes de dados
  - Tempo do GC
  - DAG
- Pode obter estatísticas de jobs concluídos
  - Extrai informações dos logs de eventos da execução
  - Exibe-as na UI do Spark

# Use Transformações Eficientes

- As escolhas de transformação afetarão muito a sua performance
  - Vimos isso em muitos exemplos
  - Algumas orientações ajudarão
- Use **DataFrames/Datasets** para obter os benefícios do Catalyst/Tungsten
- **Minimize os shuffles** e use “joins” eficientes
  - Como vimos anteriormente — O Catalyst vai te ajudar nisso
  - Considere usar variáveis de transferência e acumuladores
- Cuidado com **funções lambda** — elas podem trazer problemas para o uso do Catalyst/Tungsten
- Esteja ciente das características de seus dados
  - e.g. todos os seus dados estão em uma partição— **TEMOS UM PROBLEMA!**

# Filtre com a maior antecedência possível

- Filtre os dados não utilizados o mais cedo possível
  - Se não são processados, então jogue fora
    - Dados não utilizados causam um maior tráfego de rede, uso de CPU, requisitos de armazenamento
  - O Catalyst te ajudará com isso, mas nem sempre
    - Depende da sua transformação
- O filtro funciona em paralelo em todas as partições
  - Funções estreitas — muito eficientes
- Partições filtradas podem ser **desequilibradas**
  - Sendo uma muito maior que a outra
  - Cuidado com isso — considere o uso dos métodos coalesce ou repartition
  - Seja cauteloso — estes podem ser caros
  - Monitore o uso dos recursos (e.g. Web UI)

# Use um Bom Armazenamento de Dados

- O Spark precisa de um armazenamento de dados confiável para dados
  - Muitas opções
- **HDFS** geralmente é uma boa escolha
  - Barato, confiável e escalável
  - Infraestrutura comprovada
- **NoSQL**
  - Bom para dados real-time
  - Cassandra, Couchbase, Aerospike, etc.
  - Muitos já são integrados ao Spark
- Explorar a localização dos dados
  - Processar dados no mesmo nó - suportando alta taxa de transferência
  - Funciona bem com HDFS, pois o Spark entende seu armazenamento



# MONITORE!!!

- Coloque recursos no planejamento de seu monitoramento
  - O ajudará a diagnosticar problemas de desempenho
- Monitore o nível do SO
  - CPU, Memory
  - JVM, Garbage collection
- Monitore o nível da Aplicação
  - Tempos de transformação e uso de recursos
  - Deve ser verificado durante o desenvolvimento - geralmente pelo desenvolvedor
- Colete e represente graficamente as métricas
  - Codahale, Graphite, Grafana, Prometheus

# Não Reinvente a Roda

- O Spark possui uma comunidade ampla
- Existem muitos recursos
  - Existem vários livros excelentes de colaboradores do Spark
  - Isso é verdade até mesmo para o Spark 2
  - Use-os, siga os conselhos de seus especialistas



# Parte 7: Criando Aplicações Standalone

API Core

Construindo e executando Aplicações

Ciclo de vida de aplicações

Gerenciadores de Cluster

Logging e Debugging

# Parte 7.1: API Core

# Aplicações em Spark

- Até aqui temos usado o Spark Shell
  - Standalone
  - Ou conectando em algum cluster
- O shell é uma boa opção para
  - Operações Ad-hoc / interativas
  - Desenvolvimento rápido / Debugging
- Para códigos produtivos, temos que desenvolver uma aplicação
  - Principal diferença — você é quem cria o `SparkSession`
    - Em vez de usar uma sessão pré-criada no shell
    - Bastante simples usando algum código comum
  - Pode estar escrito em Scala / Python / Java

# Código básico de um client (Driver)

- Cria um programa (objeto com um método **main()**)
- Crie uma sessão a partir do **SparkSession.Builder**
  - Interface fluente para construção de uma sessão do Spark
  - Acesse a documentação do Builder a partir da sessão do SparkSession no site oficial do Spark

```
// É necessário importar tudo que precisamos agora
import org.apache.spark.sql.SparkSession
import org.apache.spark.sql.functions._

object MyApp {                                     // Aplicação básica (Scala)
  def main(args: Array[String]) {
    val spark = SparkSession.builder // Acessando o Builder
      .appName("MyApp")              // Definindo o nome da aplicação
      .master("local[4]")            // Execute de forma local com 4 cores
      .getOrCreate()                 // Criando a sessão
  } }
}
```

# Métodos comuns do Builder

- Todos retornam o próprio objeto Builder
  - Exceto o `getOrCreate()` que retorna a sessão

Método	Descrição	Exemplo
<code>appName(name: String)</code>	Define um nome para o app — mostrado na UI	<code>appName("MyApp")</code>
<code>master(master: String)</code>	Define a URL master do cluster para se conectar	<code>master("local[4]")</code>
<code>config(key: String, value: String)</code>	Define opções de configurações usando a chave e o valor	<code>config("cassandra.host", "host1")</code>
<code>config(key: String, value: xxx)</code>	Define uma opção de configuração com outro tipo de valor (e.g. Long)	<code>config("spark.driver.cores", 1)</code>
<code>enableHiveSupport()</code>	Habilita o suporte ao Hive	<code>enableHiveSupport()</code>
<code>getOrCreate()</code>	Obtém um <code>SparkSession</code> existente ou cria um novo (não retorna o Builder)	<code>getOrCreate()</code>

# As variantes da URL Master

Chave	Descrição	Exemplo
<b>Local</b>		
local	localhost utilizando um único core de CPU	"local"
local[N]	localhost utilizando N cores de CPU	"local[4]"
local[*]	localhost utilizando todos os cores de CPU	"local[*]"
<b>Distribuído</b>		
spark://host:port	Spark master (executando no modo Standalone)	spark://masterhost1:7077
mesos:// host:port	Spark master (executando no Mesos)	mesos://host1:5050
Yarn	Executando no YARN	"yarn"



# SparkSession vs. SparkContext

- Um `SparkSession` envolve uma instância de `SparkContext`
  - Geralmente, você programa para um `SparkSession`
  - O Spark usa o `SparkContext` empacotado internamente para processar
  - Observe que a sessão (e contexto subjacente) são um **singleton** —
    - `getOrCreate()` irá retornar uma sessão já existente
- É possível acessar o `SparkContext` conforme necessário (por exemplo, para variáveis de transmissão) via;  
**`spark.sparkContext`**
- É possível criar o `SparkContext` diretamente
  - Ao invés de criar um `SparkSession`

# Algumas Propriedades Comuns de Configuração

- São úteis para aplicações
  - Muito mais
  - Veja <https://spark.apache.org/docs/latest/configuration.html>

Propriedade	Valor Padrão	Ação
<code>spark.master</code>	(none)	URL do Master (o mesmo que <code>master()</code> )
<code>spark.app.name</code>	(none)	O nome da aplicação (o mesmo que <code>appName()</code> )
<code>spark.driver.cores</code>	1	O número de cores para o processamento do programa (apenas no modo cluster)
<code>spark.driver.maxResultSize</code>	1G	Tamanho total dos resultados serializados para ação do Spark
<code>spark.driver.memory</code>	1G	Quantidade de memória para o processo do driver (não usado no modo client)
<code>spark.local.dir</code>	/tmp	Diretório para espaço scratch do Spark
<code>spark.submit.deployMode</code>	(none)	"client" (execução local) ou "cluster" (execução em um nó do cluster)

# Configurando Propriedades para Runtime

- Pode acessar / alterar as propriedades existentes para runtime do Spark
  - Por meio do membro **SparkSession.conf**
    - Do tipo `org.apache.spark.sql.RuntimeConfig`

```
// Definindo uma única propriedade
> spark.conf.set("spark.executor.memory", "2g")

// Obtendo todas as configurações
> val configMap:Map[String, String] = spark.conf.getAll
configMap: Map[String,String] = Map(spark.driver.host -> 192.168.1.128,
spark.driver.port -> 49760, ..., spark.executor.memory -> 2g, ...)
```

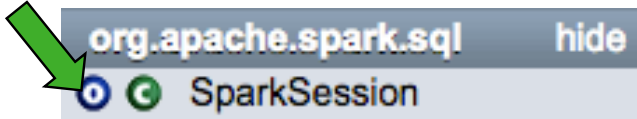
# Outras Opções de Configurações

- É possível criar uma instância de **SparkConf** e definir as suas propriedades
  - E então passa-la para o Builder a partir do método **config(conf: SparkConf)**
- É possível passar propriedades a partir do comando `spark-submit` usando **--conf**  

```
./bin/spark-submit ... \  
    --conf spark.master=spark:/1.2.3.4:7077
```
- O `spark-submit` faz a leitura de configurações no arquivo **<spark>/conf/spark-defaults.conf**
  - No formato de arquivo de propriedades chave = valor padrão
  - Ordem de precedência (da mais alta para a mais baixa):
    - (1) Propriedades definidas no Builder
    - (2) Propriedades passadas para o comando **spark-submit**
    - (3) *spark-defaults.conf*

# MINI-LAB: Reveja a Documentação

## Mini-Lab

- Acesse a documentação em <http://spark.apache.org/docs/latest/>
  - Na barra de menu superior, vá para **API Docs | Scala**
  - No painel esquerdo, digite **SparkSession** no campo de filtro
- Clique no O para ir para a documentação do Objeto
- Na documentação do método `builder()`, clique no valor de retorno do **Builder**
- Isso te levará para a documentação do **SparkSession.Builder** — reveja
- Vá até a documentação do `SparkSession`, e reveja o membro **conf**
- Clique no tipo **RuntimeConfig**, e reveja essa classe
- Acesse <https://spark.apache.org/docs/latest/configuration.html>
- Gaste alguns minutos revendo isso

## Parte 7.2: Construindo e executando aplicações

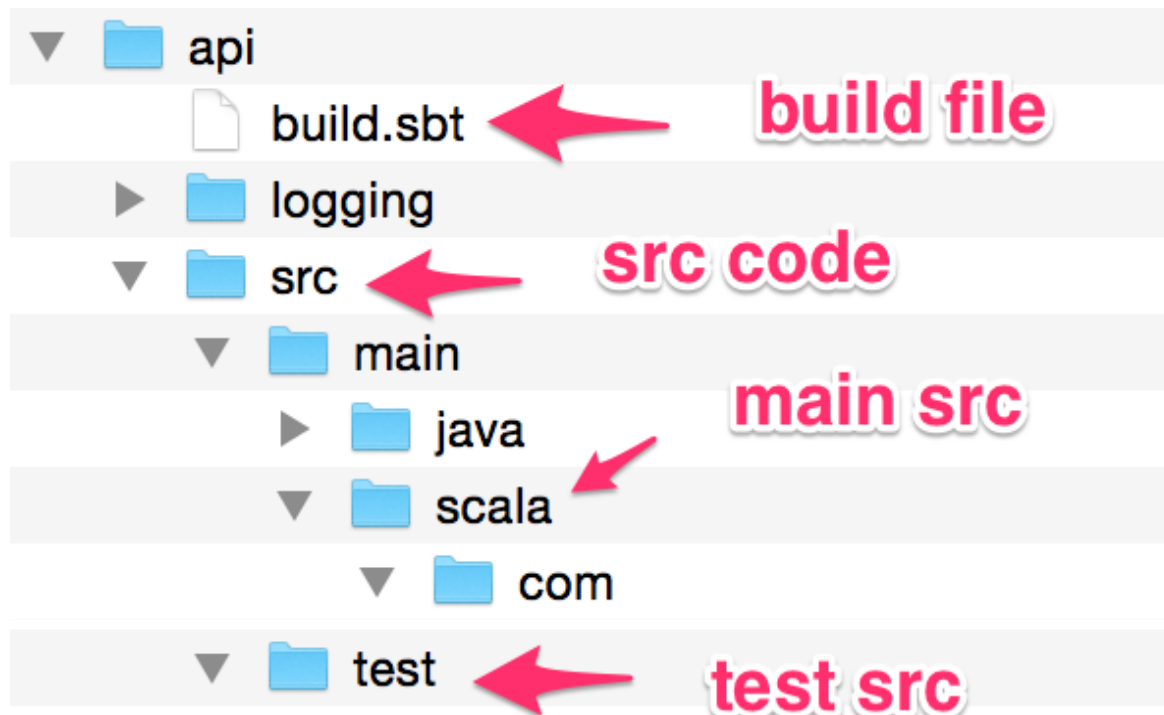
# Ferramentas para Construção/Codificação

- Existem muitas opções, vamos usar apenas o sbt
- **sbt**: Simple Building Tool (Para Scala)
  - <http://www.scala-sbt.org/>
- **maven**: Somente necessário para as dependências do Spark — por exemplo:

```
<dependency>  
  <groupId>org.apache.spark</groupId>  
  <artifactId>spark-  
core_2.11</artifactId>  
  <version>2.1.1</version>  
</dependency>
```
- **Scala IDE**: IDE do Scala baseada no Eclipse
- **IntelliJ**: Excelente suporte ao Scala, compilação rápida / incremental
- **Sublime**: Editor de texto sofisticado - suporte completo para Scala
  - <http://www.sublimetext.com/>

# Layout de uma Aplicação no sbt

- Utiliza o layout do maven por padrão





# build.sbt

- O arquivo de build do sbt
  - Este define o nome do app, a versão do app e a versão do Scala
  - Em seguida ele configura as dependências
  - Entraremos em detalhes suficientes sobre sbt para uso básico, mas não detalharemos profundamente

```
name := "MyApp"

version := "1.0"

scalaVersion := "2.11.7"

// += significa sequência concatenada de dependências
// %% significa anexar a versão Scala à próxima parte
libraryDependencies += Seq(
  "org.apache.spark" %% "spark-core" % "2.1.0" % "provided"
)

// precisa disso para trabalhar com arquivos no S3 e HDFS
// += Significa apenas adicionar a dependência
libraryDependencies += "org.apache.hadoop" % "hadoop-client" % "2.7.0"
exclude("com.google.guava", "guava")
```

# Compilando o Código

- *build.sbt* geralmente fica no diretório raiz do projeto
  - O mesmo que o *pom.xml* do Maven
- Automaticamente faz o download das dependências
- Comandos do sbt
  - sbt **compile**
  - sbt **package** — constrói um Jar
  - sbt **assembly** — constrói um “fat jar” com todas as dependências
  - sbt **clean** — exclue todos os artefatos gerados
- Pra reconstruir completamente  
**sbt clean package**
  - A primeira execução demora mais pois faz o download de todas as dependências

# spark-submit: Enviando uma Aplicação

- `<spark>/bin/spark-submit` envia o app para executar no cluster
  - Pode ser usado com todos os gerenciadores de cluster suportados
- Abaixo, submetemos a um gerenciador Standalone, configuramos a memória do executor e passamos um argumento (um nome de arquivo)

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

```
$ spark-submit --master spark://localhost:7077 \  
  --executor-memory 4G --class com.mycompany.MyApp \  
  target/scala-2.11/myapp.jar 1G.data
```

# Parâmetros do spark-submit

Parâmetro	Descrição	Exemplo
--master <master url>	URL Master	--master Spark://host1:7077
--name <app name>	Nome do app	--name MyApp
--class <main class>	Classe principal	--class com.mycompany.MyApp
--driver-memory <val>	Memória para o app (por padrão é 512M)	--driver-memory 1g
--executor-memory <val>	Memória para os executores (MAIS IMPORTANTE!)	--executor-memory 4g
--deploy-mode <deploy-mode>	Enviar o código para um worker (cluster) ou executar local (client)	--deploy-mode cluster
--conf <key>=<value>	Configuração de propriedades	
--help	Mostrar todos os comandos	

# Lab 7.1: Fazer o spark-submit de um Job