Lab 3.1: RDD básico

Detalhes do método reduce()

- **`reduce(f: (T, T) ⇒ T)`**: Reduzir elementos usando a função f()
  - `f()` processa dois elementos da RDD, retorna apenas um.
    - Por exemplo, fazendo a adição de dois números
  - **Retorno**: Um único elemento
    - f() é aplicada repetidamente, até que sobre apenar um elemento (o resultado)
- Exemplos:


```
> val numbers = sc.parallelize (List(1,2,3,4))

// Reduzir fazendo a soma de todos os elementos juntos
> numbers.reduce ( (a,b)=> a+b)
// Ou numbers.reduce(_+_)
res16: Int = 10

// Reduzir fazendo a multiplicação de todos os elementos (fatorial)
> numbers.reduce ( (a,b)=> a*b)  // Ou reduce(_*_)
res19: Int = 24
```

Detalhes do método flatMap()

- `flatMap(f: (T) ⇒ TraversableOnce[U])`: aplica a função f para todos os elementos, e então, expande o resultado
  - f retorna um objeto que pode ser iterado (e.g. uma coleção)
  - Os elementos de cada iteração são combinados e formam uma RDD ”expandida"

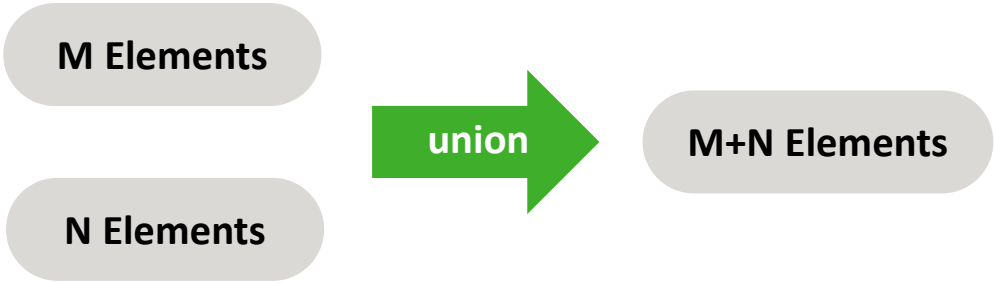

```
> val numbers = sc.parallelize (List(1,2,3))

// Mapear cada número para criar uma lista de 3 números
> val mapped = numbers.flatMap(a=> List(a-1, a, a+1))
mapped: org.apache.spark.rdd.RDD[Int] = MapPartitionsRDD[4] at flatMap at
<console>:23

> mapped.collect  // 9 elementos – 3 para cada elemento original
res14: Array[Int] = Array(0, 1, 2, 1, 2, 3, 2, 3, 4)
```

Detalhes do método union()

- `union(other: RDD[T]): RDD[T]`: Retorna a união de duas RDDs
  - Recebe como parâmetro duas RDDs
  - Dados duplicados estão incluídos (podem ser removidos com o método distinct())




```
> val odds = sc.parallelize (List(1,3,5,7))
> val evens = sc.parallelize (List(2,4,6,8))

// Criar a união
> val all = odds.union(evens)
all: org.apache.spark.rdd.RDD[Int] = UnionRDD[2] at union at <console>:25

> all.collect.sorted
res20: Array[Int] = Array(1, 2, 3, 4, 5, 6, 7, 8)
```

Resumo de Transformações em RDD

RDD r = {1,2,3,3}

Transformação	Descrição	Exemplo	Resultado
map(func)	Aplicar uma função em cada elemento	r.map(x => x*2)	{2,4,6,6}
filter(func)	Filtra todos os elementos e retorna apenas os que possuem o resultado da função = true	r.filter(x=> x % 2 == 1)	{1,3,3}
distinct	Remover dados duplicados	r.distinct()	{1,2,3}
flatMap	Funciona como o Map, mas um elemento pode retornar mais de um resultado		
mapPartitions	Funciona como o Map, mas executa em toda a partição, não em cada elemento		

Resumo de Transformações em RDD

RDD r1 = {1,2,3,3} **RDD r2 = {2,4}**

Transformação	Descrição	Exemplo	Resultado
union(RDD)	Une dois RDDs (dados duplicados estão incluídos)	r1.union(r2)	{1,2,3,3,2,4}
intersection (RDD)	Retorna elementos em comum entre dois RDDs	r1.intersection(r2)	{2}
subtract(RDD)	Tira os elementos em comum entre dois RDDs	r1.subtract(r2)	{1,3,3}
sample	Obtém uma amostra dos dados de uma RDD		

Resumo de Ações em RDD

- Ações retornam valores, **não RDDs**
  - e.g. o método count() retorna um Long
  - **Acionam as execuções do DAG**

RDD r = {1,2,3,3}

Action	Description	Example	Result
count()	Conta todos os registros de uma RDD	r.count()	4
first()	Obtem o primeiro registro de uma RDD	r.first ()	1
take(n)	Obtem as primeiras N linhas	r.take(3)	[1,2,3]
collect()	Coletar todos os registros de uma RDD Isso que dizer que todos os dados vão para a memória de uma única máquina, então **MUITO CUIDADO!**	r.collect()	[1,2,3,3]
saveAsTextFile()	Salvar em um File System		
	… Muito mais— veja a documentação…		

Transformação Mais Complexa

- A API do RDD tem consideravelmente mais capacidade
  - Em particular, operações em pares de chave/valor
  - Estes são geralmente usados para agregação (agrupamento, contagem, etc.)
- Vamos primeiramente apresentar a API de DataFrame/Dataset
  - Possue uma interface simples
- Iremos introduzir operações mais complexas
  - Focando na API de DataFrame/Dataset
  - Fornecendo alguns exemplos de programação com a API do RDD

LAB

MINI-LAB: Reveja a documentação do RDD

- Você encontra a documentação no site do Apache Spark

Mini-Lab

- Busque por http://spark.apache.org/docs/latest/
  - Na barra superior, clique em "**API Docs | Scala**"
  - No painel esquerdo, encontre o pacote **`org.apache.spark.rdd`**
    - Dica: Clique em "**display packages only**" para recolher a lista de pacotes
    - Procure por `org.apache.spark.rdd` e clique em "**Show**" para expandir
    - Dentro desse pacote, clique na entrada RDD
    - Como alternativa, pode pesquisar por RDD no filtro de pesquisa no canto superior esquerdo
    - Isso te leva para a documentação da API do RDD
- Reveja a API do RDD brevemente
  - Em especial, veja os métodos `map()`, `filter()`, `count()`, and `take()`

Lab 3.2: RDD na prática

Perguntas de Revisão

- O que é RDD?
- O que acontece quando você executa uma transformação em uma RDD?
- Onde ficam alocados os dados de uma RDD?
- Quais são algumas das transformações típicas usadas com RDDs?

Resumo

- **RDD: Resilient Distributed Dataset**
  - Coleção de elementos distribuídos e particionados pelos nós do cluster
  - Abstração de dados do Spark Core

- RDDs suportam **transformações** e **ações**
  - **Transformações** criam uma nova RDD a partir de outra RDD (e.g. Mapear dados)
    - São executados no modelo lazy – criam um DAG que executa apenas quando ocorre uma ação nessa nova RDD
  - **Ações** (e.g. Coletar) extraem informações — acionam a execução do DAG

Resumo

- Os dados de um RDD são **particionados** entre um Cluster Spark
  - Cada nó possue uma parte do dado, e cada task de cada nó processa apenas o que há nesse nó

- Operações comuns em um RDD:
  - `filter`: Filtrar os elementos de um RDD
  - `map`: Aplicar uma função em cada element de um RDD
  - `collect`: Coletar todos os dados.

Parte 4: Spark SQL, DataFrames, e Datasets

- Introdução
- SparkSession e Carregamento/Armazenamento de dados
- Introdução à DataFrame/Dataset
- Query DSL
- Datasets
- Métodos flatMap, explode, split

Scala Systems

Parte 4.1: Introdução

Limitações do RDD

- API de baixo nível
  - Você específica o “Como” não o “O que”
  - A API possui pouca habilidade para lidar com formatos de dados comuns
    - e.g. JSON
- Opaco para o Spark (usa lambdas arbitrárias)
  - As consultas não podem ser facilmente otimizadas pelo Spark
  - Não é difícil escrever transformações ineficientes
- Sem suporte para consultas semelhantes a SQL
  - Limita a aplicabilidade
  - SQL é bem conhecido

Introdução ao Spark SQL

- Componente responsável pelo processamento de **dados estruturados**
  - Construído com base no Spark Core
- Suporta **schema/estrutura** para os dados no Spark
  - Estruturados como tabelas/linhas/colunas
  - **Carrega dados** de muitos tipos de origens
- Serve uma **API de alto nível** para processamento dos dados
  - **`Dataset/DataFrame`**
  - **SQL**
- **Otimiza** a execução, visando aumentar a performance
  - **Catalyst**: Otimizador de queries
  - **Tungsten**: Otimizador de uso de CPU/Memória
  - Desempenho estável entre as API's (Scala, Python, R, Java...)

Formatos de dados suportados

- Pode carregar dados de diferentes tipos de origens
  - Por padrão já suporta muitos tipos de dados
  - Muitos outros padrões podem ser suportados com o uso de bibliotecas externas
  - Pode inferir a estrutura do schema para alguns formatos

Scala Systems

DataFrames e Datasets

- Coleções de dados distribuídos (como as RDDs)
  - **`DataFrame`**: Adiciona informações de **schema**
  - **`Dataset`**: Possue schema, e adiciona **segurança de tipo em tempo de compilação**
- **Query DSL** (Domain Specific Language)
  - API abrangente de consultas — `filter`, `map`, `groupBy`, etc.
  - Interface simples e muito fluente
- Consultas em **SQL** são 100% suportadas
  - Sintaxe familiar para os desenvolvedores
  - Necessita de mais recurso em alguns casos
    - e.g. processar muitas agregações em uma única consulta

Exemplo de processamento simples

- Considere um arquivo com dados de pessoas
  - Nome, gênero, e idade
  - Devemos encontrar a **media da idade** para cada **gênero**?

- Nós vamos ilustrar isso usando RDDs e DataFrames
  - RDDs vão usar um arquivo TXT (indicado abaixo)
  - DataFrames vão usar um arquivo JSON (indicado abaixo)

```
John M 35
Jane F 40
Mike M 20
Sue F 52
```

```
{"name": "John", "gender": "M", "age": 35 }
{"name": "Jane", "gender": "F", "age": 40 }
{"name": "Mike", "gender": "M", "age": 20 }
{"name": "Sue",  "gender": "F", "age": 52 }
```

Exemplo de processamento com RDDs

- Abaixo, nós calculamos a média de idades com RDDs
  - Não é a única forma, e também não é a melhor
- Mas, podemos identificar algumas “dores”:
  - Implementação complexa
    - Pelo fato de que o RDD não infere o tipo e nem a estrutura do dado, temos que fazer isso.

```
> val folksRDD=sc.textFile("people.txt")  // Obter o dado
> folksRDD.map(_.split(" "))              // Separar isso em campos
   .map(x => (x(1), Array(x(2).toDouble, 1)))  // Emparelhar os dados
   .reduceByKey( (x,y) => Array(x(0)+y(0), x(1)+y(1)) ) // Contar
   .map(x => Array(x._1, x._2(0)/x._2(1)))  // Calcular a média
   .collect                                 // Obter os resultados

res1: Array[Array[Any]] = Array(Array(F, 46.0), Array(M, 27.5))
```

Exemplo de processamento com DataFrame

- Abaixo, nós calculamos a média usando um DataFrame

- Percebeu a diferença?
  - Implementação muito mais simples
  - O Spark **entende os detalhes dos dados** e pode otimizar

```
> val folksDF=spark.read.json("people.json") // Obter o dado
> folksDF.groupBy($"gender")                 // Agrupar por gênero
   .agg(avg($"age"))                 // Obter a média da idade de cada grupo
   .show                             // Mostrar os resultados

+------+--------+
|gender|avg(age)|
+------+--------+
|     F|    46.0|
|     M|    27.5|
+------+--------+
```

Exemplo de processamento com SQL/DataFrame

- Abaixo, calculamos a média usando `DataFrames` e SQL
- Mais simples até mesmo que o exemplo anterior
  - Mais pessoas entendem SQL
  - O Spark **entende os detalhes dos dados** e pode otimizar

```
> val folksDF=spark.read.json("people.json") // Obter o dado
> folksDF.createOrReplaceTempView("people")  // Configura para que seja
                                             // consumido via SQL

// Obtem a média por gênero
> spark.sql("SELECT gender, avg(age) FROM people GROUP BY gender").show
+------+--------+
|gender|avg(age)|
+------+--------+
|     F|    46.0|
|     M|    27.5|
+------+--------+
```

Exemplo de processamento com DataSet

- Abaixo, calculamos a média usando DataSets
  - Parecido com DataFrames — mas com segurança de tipo
  - Usa função lambda para agrupar

```
> val folksDF=spark.read.json("people.json") // Obter o dado

// Declarar uma classe para representar o dado
> case class Person (name: String, gender: String, age: Long)

// Utilizar a classe
> val folksDS=folksDF.as[Person]
> folksDS: org.apache.spark.sql.Dataset[Person] = [age: bigint... ]

// Obter a média dos grupos
> folksDS.groupByKey(T => T.gender).agg(avg($"age").as[Double]).show
+-----+--------+
|value|avg(age)|
+-----+--------+
|    F|    46.0|
|    M|    27.5|
+-----+--------+
```

O otimizador Catalyst

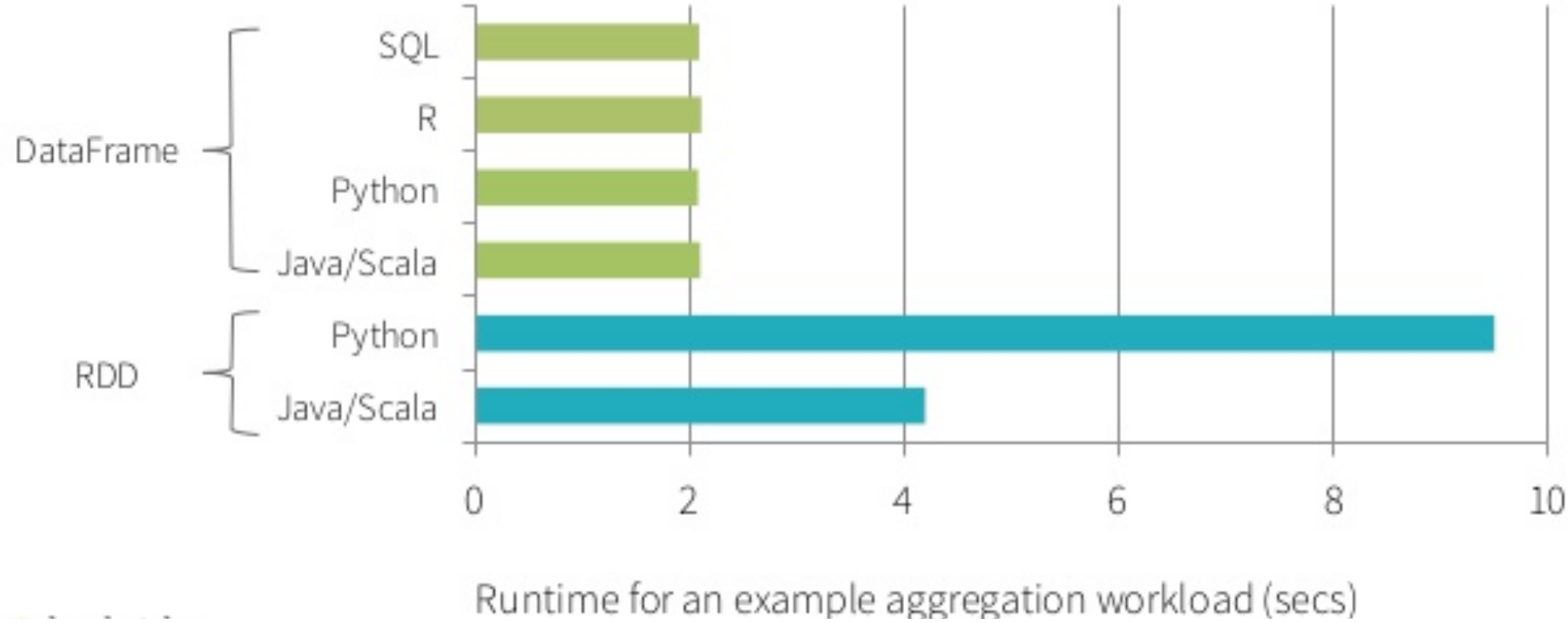
- **Otimizador de queries** em dados estruturados
  - Trabalha com `Dataset`, `DataFrame`, e SQL
  - Automáticamente transforma as queries visando aumentar a performance

- Fácilmente extendido para suportar:
  - Novas origens de dados semi-estruturados (e.g JSON)
  - Fontes de dados ”inteligentes" na qual é possível enviar filtros (e.g. HBase)
  - Funções e tipos definidos pelo client

- O Catalyst infere o schema do dado e a querie
  - Fornecendo muitas oportunidades de otimização
  - Operações “lazily evaluated”, ou seja, a sequência completa pode ser analisada para otimizações
  - **Lambdas não** podem ser analisadas (elas são opacas)

Performance altamente melhorada

- A performance do DataFrame é muito melhor que o RDD
- Performance parecida entre as linguagens (e.g. Scala and Python)
  - Elas criam planos de execução parecidos

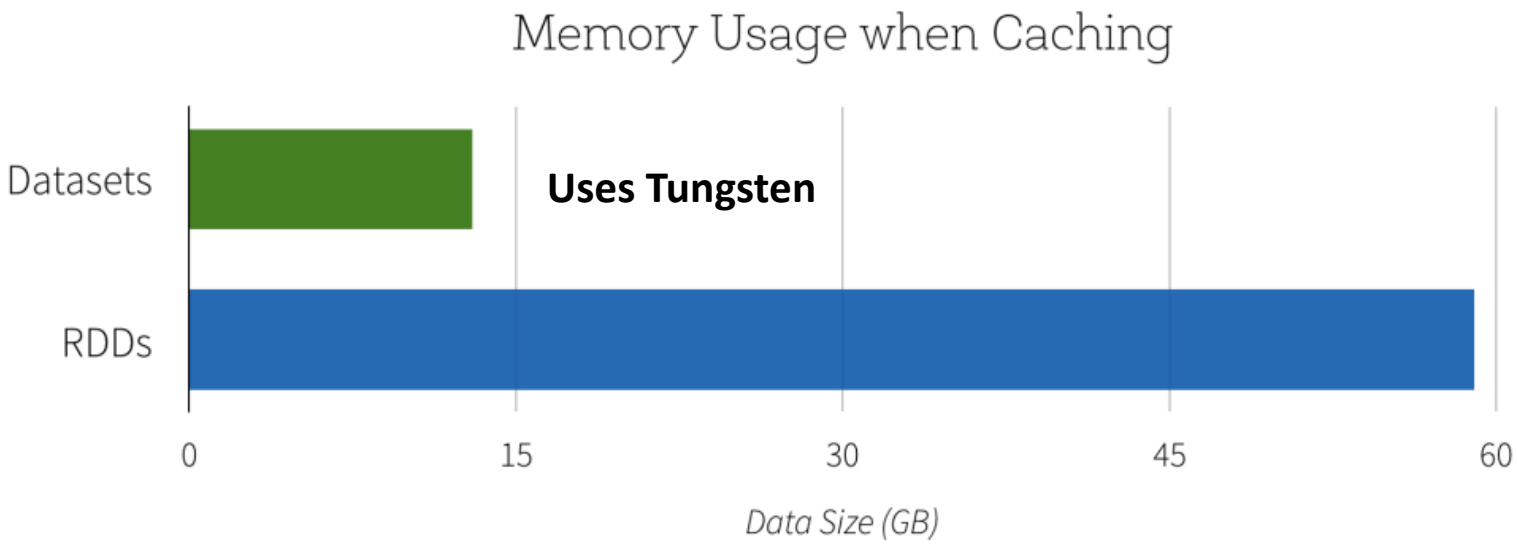


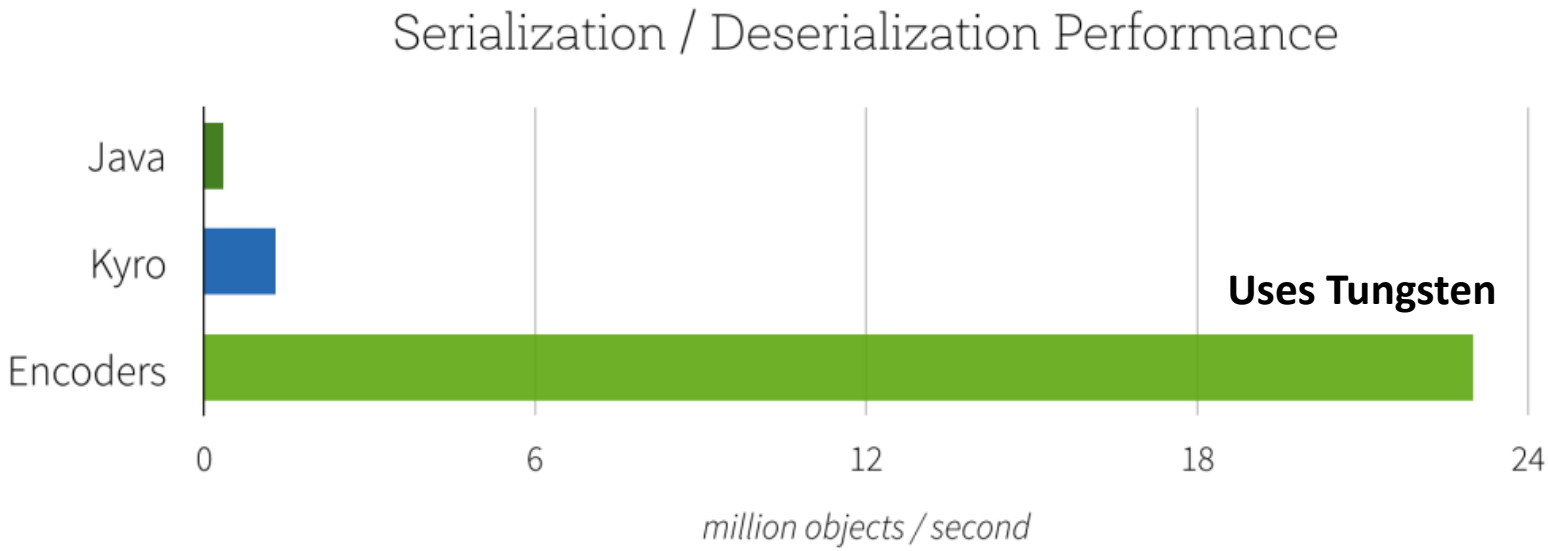
Scala Systems

O otimizador Tungsten

- Melhora consideravelmente a **performance de CPU/Memória** do Spark
  - CPU/memory estão cada vez mais se tornando gargalos
    - I/O, performance de rede, e armazenamento de dados são bons no Spark
  - O desempenho fica mais perto dos limites de hardware
- Armazena dados em memória em formato binário em modo **”off-heap”**
  - Diferente do modo de armazenamento de objetos em Java(on-heap)
  - Reduz o uso da memória “heap” e elimina o Garbage Collector (GC)
  - Pode operar diretamente com o objeto binário (sem desserialização)
  - Compreende e otimiza em diferentes caches (L1, L2, …)
- **Gera códigos** para avaliação de expressão
  - Sem a necessidade de desserialização para muitas operações

Memória reduzida / Performance aumentada

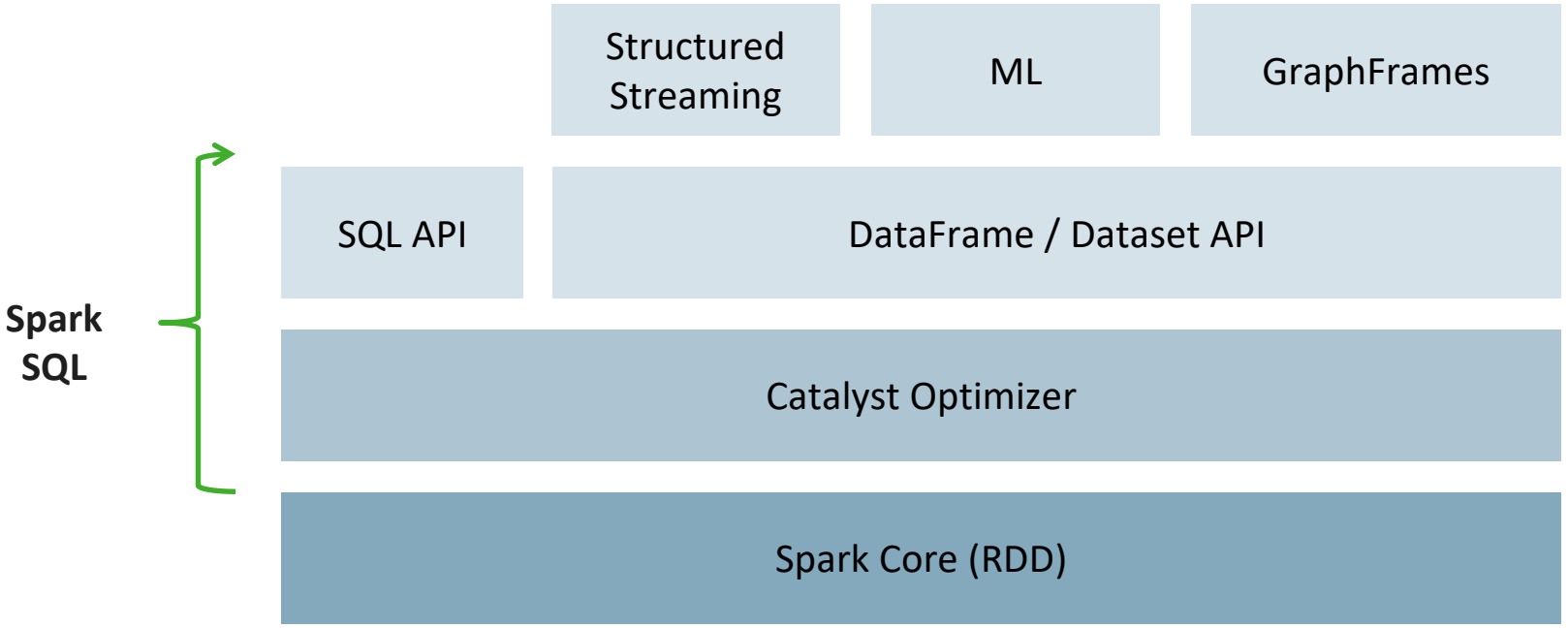




Scala Systems

Estrutura do Spark com DataFrames/Datasets

- Spark SQL é um bloco de construção central
  - Suporta otimizações do Catalyst para outros módulos
  - Outros módulos estão evoluindo
    - GraphX (baseado em RDD) => GraphFrames (baseado em DataFrame)



Resumo

- Spark SQL: API de alto nível para processamento em Spark
  - Trabalha com dados estruturados (e semi-estruturados)
- Possue os componentes:
  - API DataFrame/Dataset
  - API SQL
  - Catalyst (otimizador de query)
  - Tungsten (otimizador de processamento)
- Se tornando a **interface principal** para desenvolvimento Spark
  - Novos desenvolvimentos baseados no uso de DataFrame/Dataset
    - Incluindo bibliotecas de níveis mais altos, como o processamento em grafos (GraphFrames)
  - APIs continuam evoluindo — ainda existe a necessidade de RDDs para alguns casos

Parte 4.2: SparkSession e Carregamento/Armazenamento de dados

Scala Systems

Principais tipos de API

- Todos os tipos falados até agora estão em: **`org.apache.spark.sql`**
- **`DataFrame`**: Coleção distribuída com um **schema**
  - Sinônimo para **`Dataset[Row]`**
- **Dataset:** Coleção **fortemente tipada** com schema
- **`Column`**: Uma coluna em um DataFrame
  - Usado para criar expressões no Query DSL
- **`Row`**: Representa o dado no formato tabular
- **`SparkSession`**: Ponto de entrada para o Spark SQL
  - `Substitui os antigos objetos SQLContext e HiveContext`
  - Pré-criados no Spark shell
- **`DataFrameReader/Writer`**: Para carregar/armazenar dados

SparkSession (ponto de entrada da API)

- As habilidades da classe incluem:
  - Ler arquivos de dados (via **`DataFrameReader`**)
  - Criar instâncias de DataFrame e Dataset
  - Executar consultas SQL

- Instâncias acessadas a partir de um Builder (uma Fábrica)
  - Builder é parte do **objeto** SparkSession
  - Mais detalhes em breve
  - Uma sessão é pré-criada no REPL na variável **`spark`**
    - `getOrCreate()` para criar ou retornar uma instância já criada

```
> SparkSession.builder.getOrCreate
org.apache.spark.sql.SparkSession =
org.apache.spark.sql.SparkSession@748321c5

> spark
org.apache.spark.sql.SparkSession =
org.apache.spark.sql.SparkSession@748321c5
```

DataFrameReader

- Interface para carregar dados de uma origem externa
  - Obtida via **`SparkSession.read()`**
- Por padrão já suporta leitura de formatos de dados comuns
  - **Infere o schema** automáticamente
  - json, parquet, csv, jdbc para databases relacionais, hive, entre outros
- Abaixo, o método **`spark.read()`** retorna um `DataFrameReader`
  - **`json("people.json")`** carrega dados do arquivo *people.json*
    - O dado precisa estar no formato de linhas JSON
    - Um objeto JSON por linha, mais algumas outras limitações

```
val folksDF=spark.read.json("people.json") // Obter o dado
```

DataFrameReader API

- Métodos para carregar arquivos em formatos específicos
  - `csv(path: String)`: Carrega um arquivo no formato CSV
  - `jdbc(…)`: Carrega dados de um database relacional
  - `json(path: String)`: Carrega um arquivo no formato JSON
  - `parquet(path: String)`: Carrega um arquivo no formato PARQUET
  - `text(path: String)`: Carrega um arquivo no formato TEXTO
  - Entre outros

- Pode-se também explicitar detalhes do formato dos dados
  - `format()`: Específicar o formato do dado
    - Recebe uma classe ou um nome de um formato (e.g json)
  - `schema()`: Específicar o schema do dado
    - Detalhes do dado (via instâncias `StructType/StructField`)

```
val folksDF=spark.read.format("json").schema(…).load("people.json")
```

DataFrameWriter

- Interface para armazenar dados em uma fonte externa
  - Obtido via **`SparkSession.write()`**
  - Habilidades similares ao `DataFrameReader`
    - Por padrão suporta escritas nos formatos csv, jdbc, json, parquet, e texto
- Abaixo, exemplificamos o armazenamento de dados no formato parquet
  - O dado foi lido no formato JSON
  - Pode-se fácilmente transformar o dado (JSON => parquet)

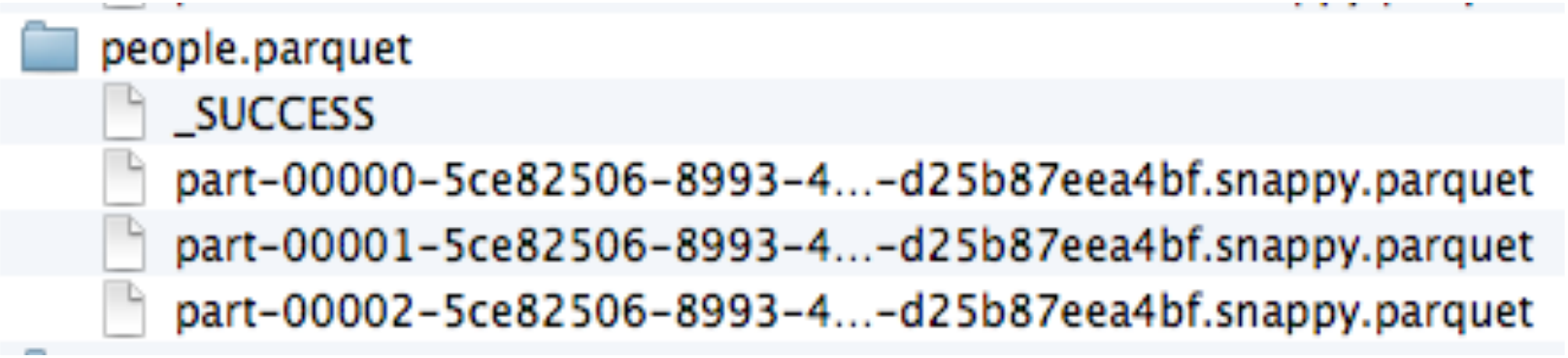
```
val folksDF=spark.read.json("people.json") // Obter o dado (JSON)

folksDF.write.parquet("people.parquet")    // Escrever o dado (parquet)
```

Múltiplos arquivos de escritas

- **O formato de armazenamento** para a escrita de um DataFrame é:
  - **Pasta para armazenamento** com o nome escolhido
  - **Múltiplos arquivos** com os dados dento da pasta
    - Um arquivo para cada partição, que geralmente vai ser escrito para um File System distribuído (e.g. HFS)
    - É possível escrever tudo em um único arquivo

- Abaixo, um exemplo de escrita para o *people.parquet*
  - 3 partições



Scala Systems

Interfaces fluentes

- **Interfaces fluentes** foram criadas para tornar o código mais legível e fluido
  - Torna simples a leitura e a escrita
  - Geralmente permite fazer cadeias de chamadas e as vezes usam Builders
- A API do Spark SQL utiliza Interfaces Fluentes
  - Abaixo, um exemplo de Interface Fluente na API do Spark SQL e outro exemplo do não uso de uma Interface Fluente para o mesmo caso

```
val folksDF=spark.read.format("json").schema(…).load("people.json")
```

```
// Exemplo não fluente
val reader = spark.read
reader.setFormat("json")
reader.setSchema(…)
val folksDF=reader.load("people.json")
```

LAB

MINI-LAB — Reveja a Documentação

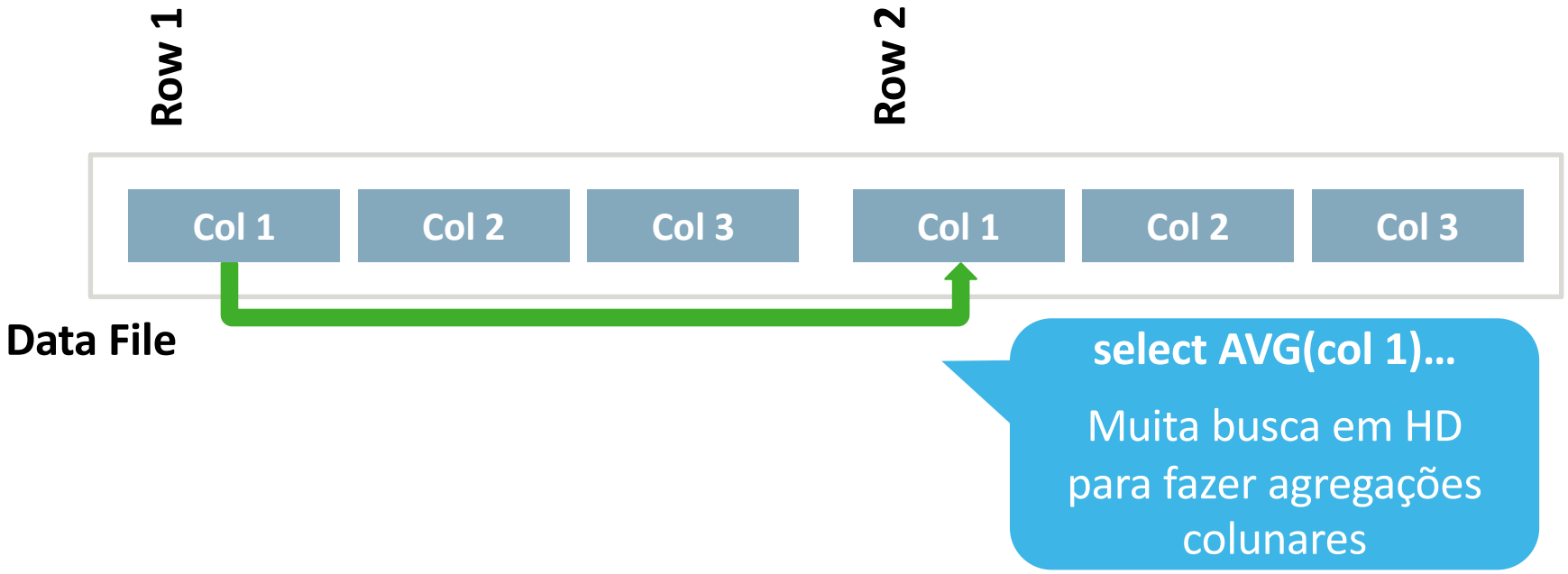
- Acesse a documentação do Spark em: http://spark.apache.org/docs/latest/
  - Busque por `org.apache.spark.sql` no paínel esquerdo
- Reveja:
  - `Classe SparkSession` (clique em `C` perto dele na lista)
    - Reveja os métodos `read()` e `createDataFrame()`
  - `Objeto SparkSession` (clique em `O` perto dele na lista)
    - Veja os métodos
    - Siga o link para o `Builder` da classe e reveja-o
  - `DataFrameReader` e `DataFrameWriter`
    - Reveja os métodos que aprendemos nessa aula

Formatos de dados

- O Spark suporta muitos formatos de dados
- Vamos passar por uma breve visão geral dos formatos suportados
- Os formatos suportados:
  - Formatos baseados em linhas e colunas
  - Formatos baseados em Texto (e.g. JSON and CSV)
  - Formatos Binários

Visão geral: Armazenamento baseado em linhas

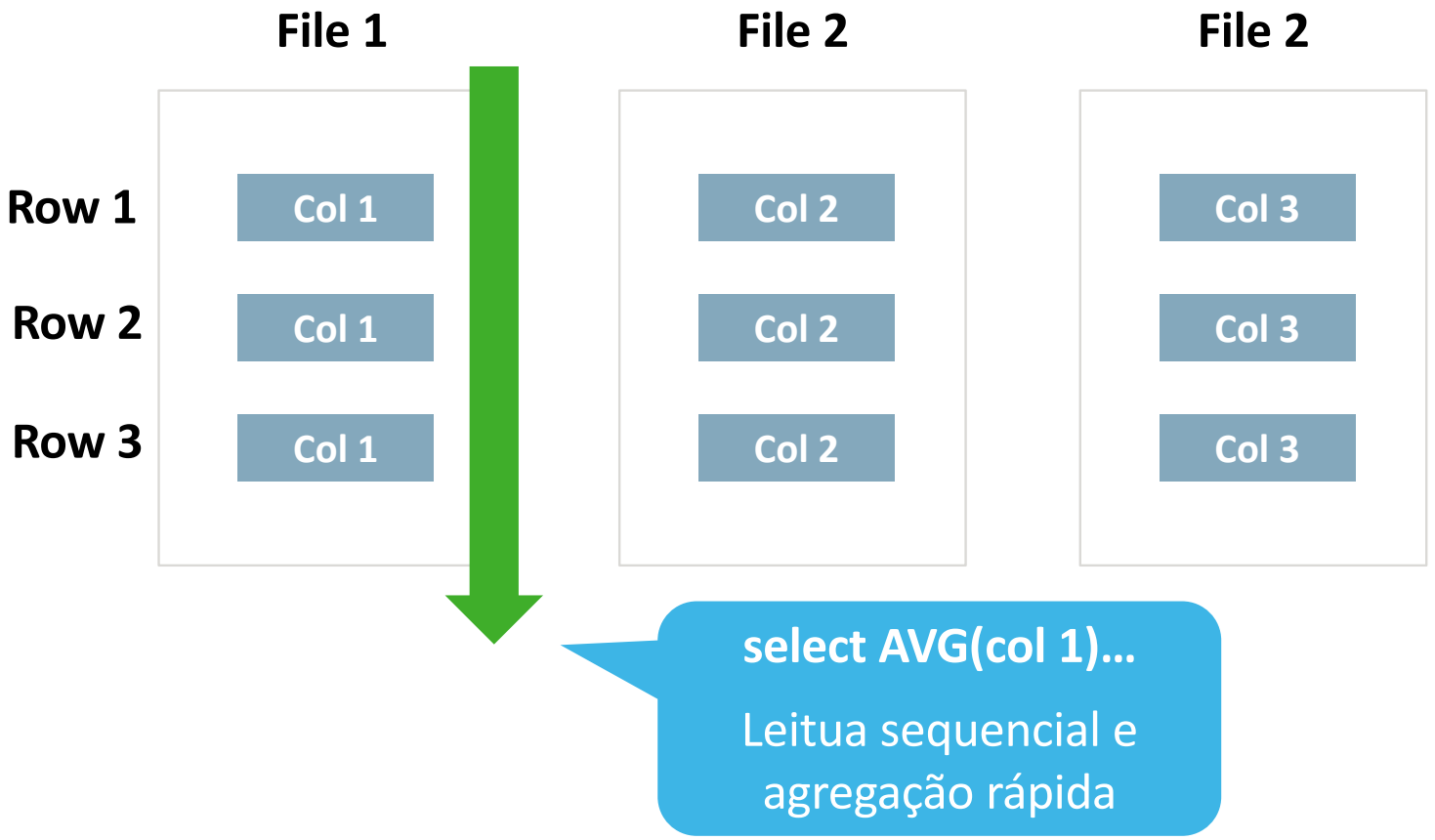
- Linhas armazenada fisicamente juntas
  - Geralmente bastante indexado (e.g. DB relacional)
  - Uma boa escolha para consultas do tipo “select *”
  - Não é uma boa escolha para agregação (e.g. calcular média)



Scala Systems

Visão geral: Armazenamento baseado em colunas

- Armazenamento de **colunas** fisicamente juntas
  - Otimizado para agregação de uma única coluna
    `select MAX(temp) from sensors;`
  - Não é uma boa escolha para buscas do tipo "`select *`"



Scala Systems

Formatos de dados comuns baseados em texto

- Todos são baseados em linhas
- **JSON**: (JavaScript Object Notation)
  - Formato leve de transferência de dados
  - Leitura via `DataFrameReader.json()`
    - Automáticamente **infere** o schema
- **CSV**: (Comma Separated Values)
  - Formato simples de dados tabulares
  - Leitura via `DataFrameReader.csv()`
    - Autimáticamente **infere** o schema
- Formato de texto live
  - Leitura via `DataFrameReader.text()`
  - Geralmente analisa o texto e aplica um schema manualmente

Parquet

- Formato de dados colunares (um projeto Apache)
  - Armazenamento baseado em binários, compactação efiente
  - O schema é armazenado junto do arquivo (o arquivo é **autodescritivo**)
  - Muito eficiente para consultas colunares
  - Possui bom suporte no ecossistema Hadoop e em outras ferramentas
- Escolhido como formato padrão em muitos lugares
- Leitura via `DataFrameReader.parquet()`

Outros formatos

- **Avro**: Formato binário baseado em linhas (projeto Apache Avro)
  - O schema é armazenado junto de parte do dado
  - Suporta versionamento de schema
  - Suportado pela biblioteca spark-avro externa do Spark
- Formatos **baseados no Hadoop** (binário / sequencial)
  - Baseados em linhas, pares de chave/valor
  - Suportado por métodos do SparkContext (`binaryFile, hadoopFile, newAPIHadoopFile`)
  - Baseados em RDD
- **Optimized Row Columnar** (ORC): Formato hibrido de linhas e colunas
  - Armazena linhas, e dentro de linhas possue-se dados armazenados em formato colunar
  - Comum encontrar em armazenamento de dados do Hive
  - O Spark pode suportar usando as bibliotecas do Hive