

# 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

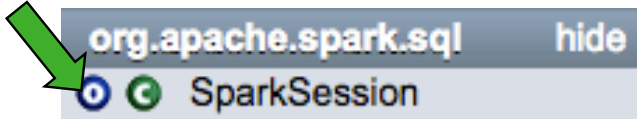
- É possível criar uma instância de **SparkConf** e definir as suas propriedades
  - E então passa-la para o Builder a partir do método **config(conf: SparkConf)**
- É possível passar propriedades a partir do comando `spark-submit` usando **--conf**  

```
./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

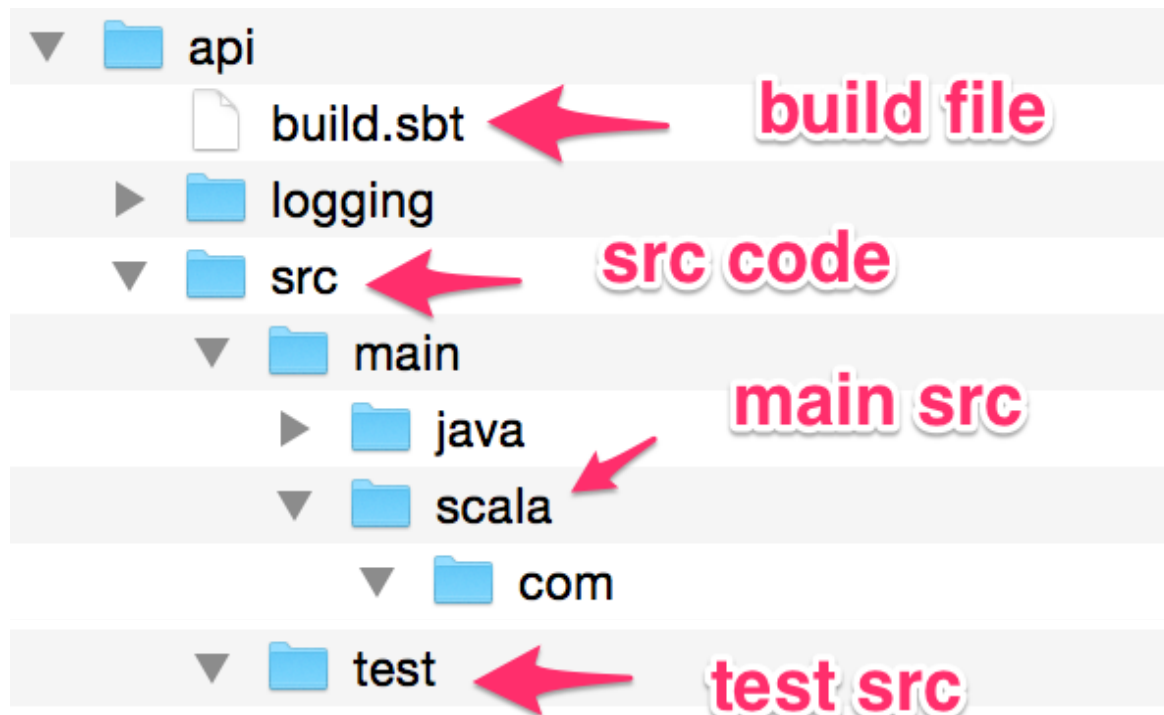
# Ferramentas para Construção/Codificação

- 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

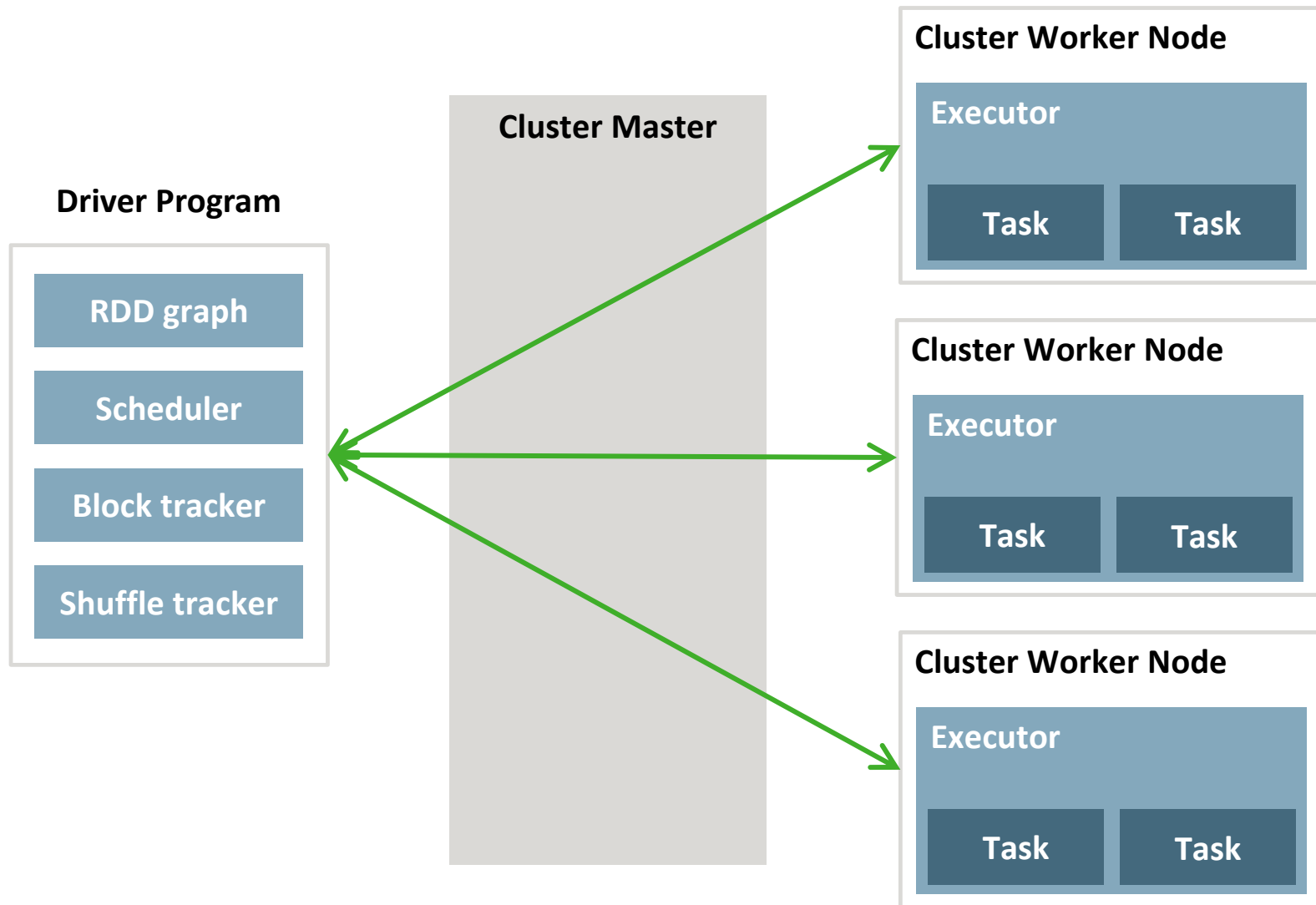
Parâmetro	Descrição	Exemplo
--master <master url>	URL Master	--master Spark://host1:7077
--name <app name>	Nome do app	--name MyApp
--class <main class>	Classe principal	--class com.mycompany.MyApp
--driver-memory <val>	Memória para o app (por padrão é 512M)	--driver-memory 1g
--executor-memory <val>	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 (Faremos juntos)

# Ciclo de vida da Aplicação

# Arquitetura de uma Aplicação Spark



# A Aplicação Driver (o Client)

- O método “main” de uma aplicação
  - É onde o **SparkSession/SparkContext** é criado
  - Estabelece uma conexão com o cluster
  - Cria o DAG (Direct Acyclic Graph) de operações
- Conecta ao gerenciados de recursos do cluster
  - Obtém **executors** em nós workers
  - Envia os códigos do app para os executors
  - Envia **tasks** para os executors
- O driver deve estar próximo aos nós workers
  - De preferência na mesma LAN

# Executors e Tasks

- **Executors**

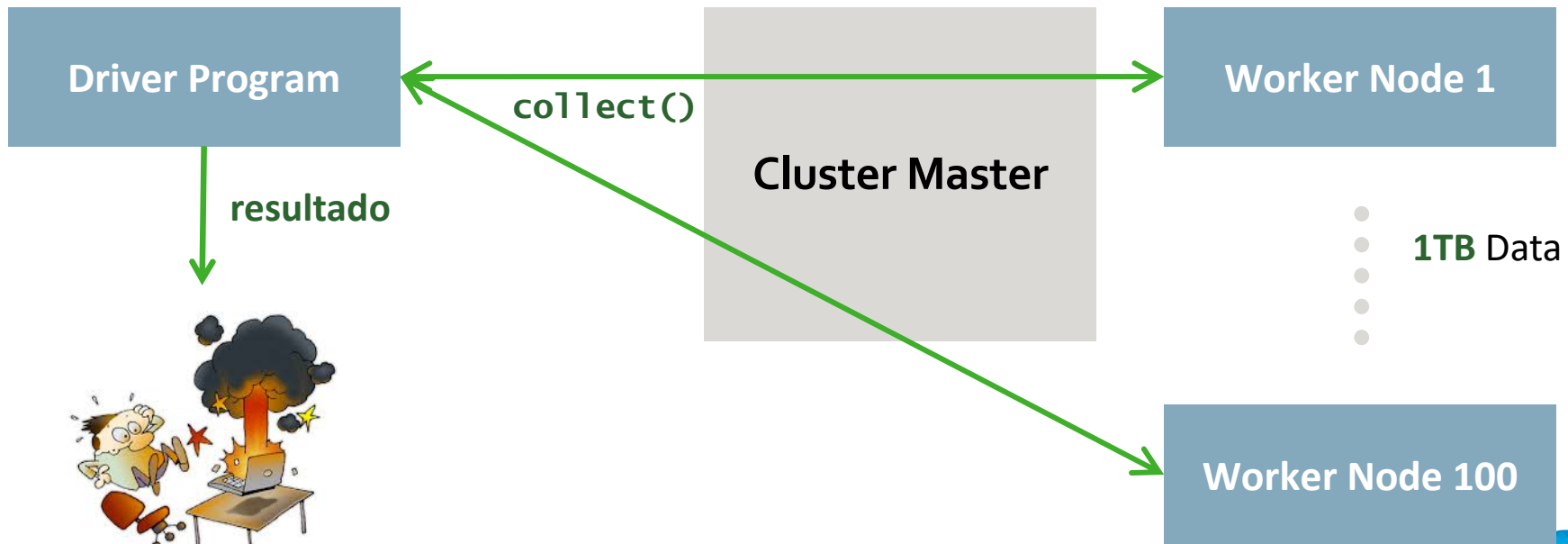
- Processos que executam processos e armazenam dados
- Cada app possui o seu executor
- Criado na inicialização do app, processa até a duração do app
- Containers JVM (tarefas de diferentes aplicativos em diferentes JVM)
- Executam Tasks (em threads)
- Fornece memória para armazenamento em cache

- **Tasks**

- “Menores” unidades de execução
- Processam dados em partições
- Leva em consideração a “localidade dos dados”
- É executado como “thread” na JVM do executor

# Memória do Driver vs. Memória do Executor

- A memória do driver geralmente é pequena
- A memória do executor é onde os dados são armazenados — pode ser grande
- **RDD.collect()** ou operações similares enviam dados para o driver
  - Grandes coleções podem causar uma sobrecarga de memória do driver
  - Busque por uma alternativa melhor!



## Parte 7.4: Gerenciadores de Cluster

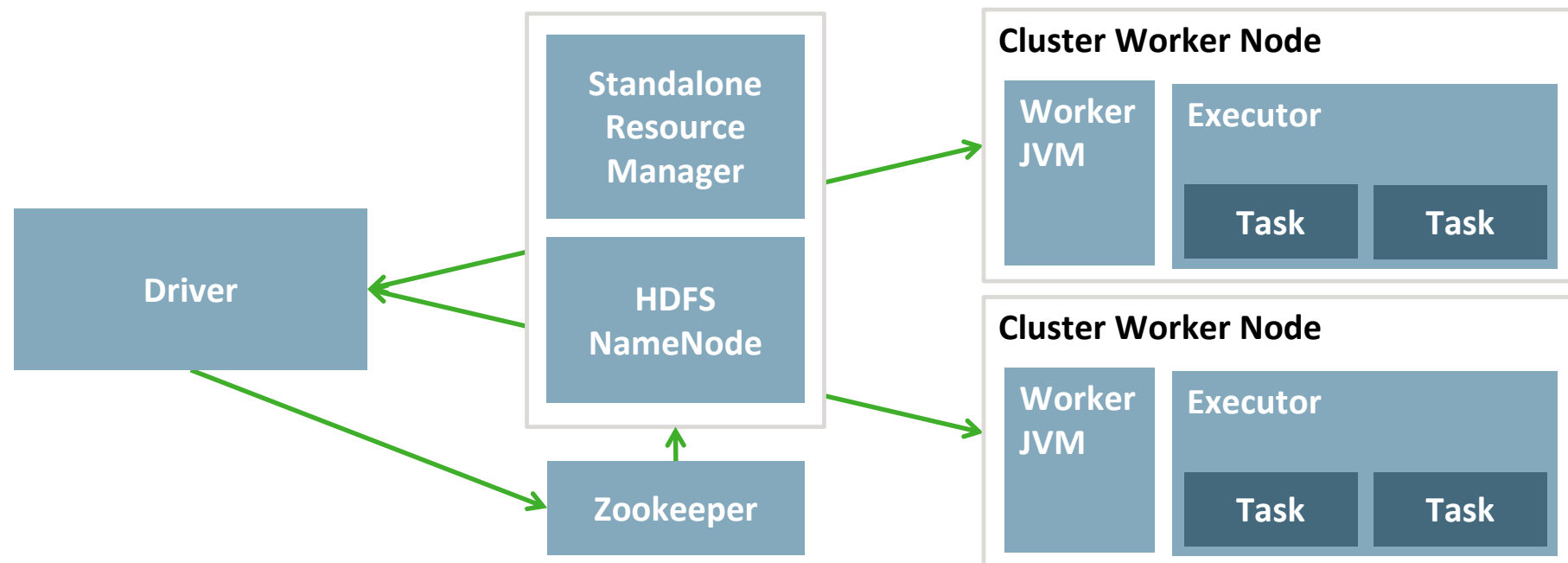
# Visão geral sobre gerenciadores de Cluster

- **Standalone**: Gerenciador de cluster exclusivo do Spark
  - Para enviar aplicativos a um cluster autônomo, use uma URL Master no formato **spark://<master-node>:7077**
  - É possível acessar a UI de Gerenciamento no endereço **http://<master-node>:8080**
- Apache **YARN**: Gerenciador de cluster Hadoop / MR
  - **Y**et **A**nother **R**esource **N**egotiator
  - Oferece gerenciamento e agendamento de recursos
  - É desacoplado do processamento de dados
- Apache **Mesos**: Gerenciador de cluster criado em UC Berkeley
  - Por algumas das mesmas pessoas que criaram o Spark
  - Fornece alocação dinâmica de recursos para várias estruturas



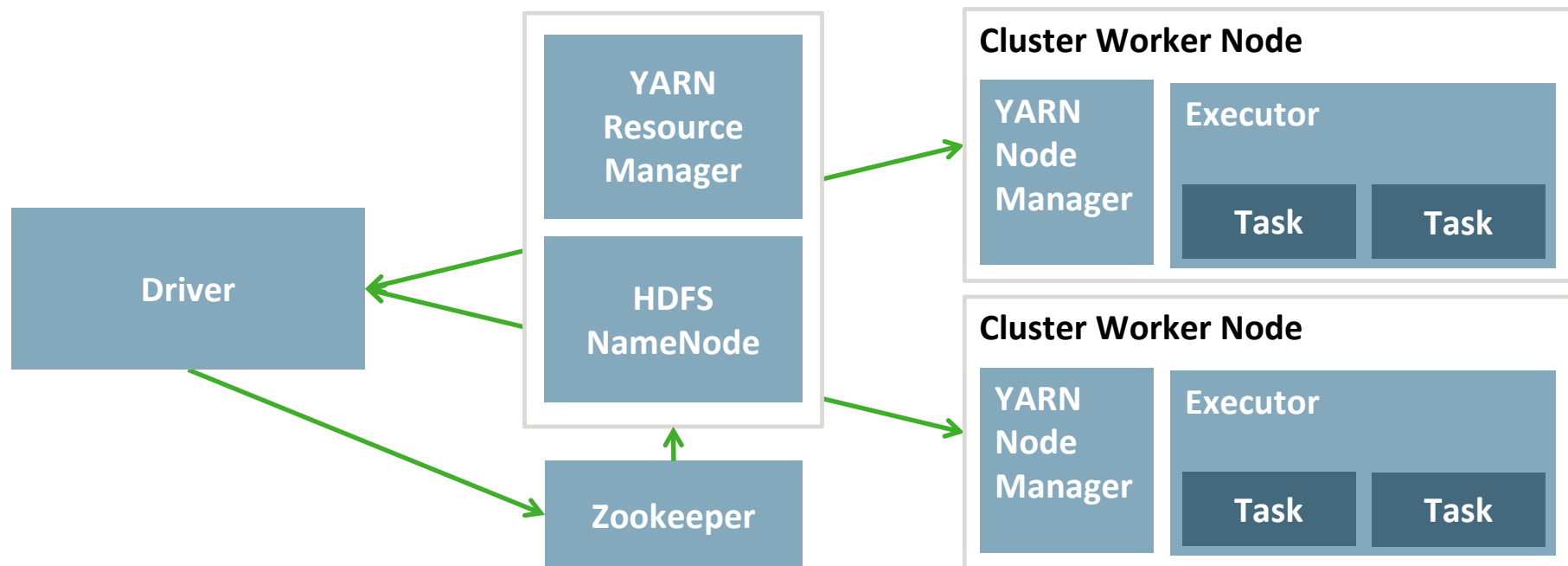
# Spark na Arquitetura Standalone

- O gerenciador Standalone é parte do Spark
- JVMs Workers iniciam os Executors
- Adequado para muitos sistemas de produção
- Tolerância a falhas na Master a partir do Zookeeper



# Spark na Arquitetura YARN

- **YARN Node Manager** inicia os Executors
- Tolerância a falhas na Master a partir do Zookeeper



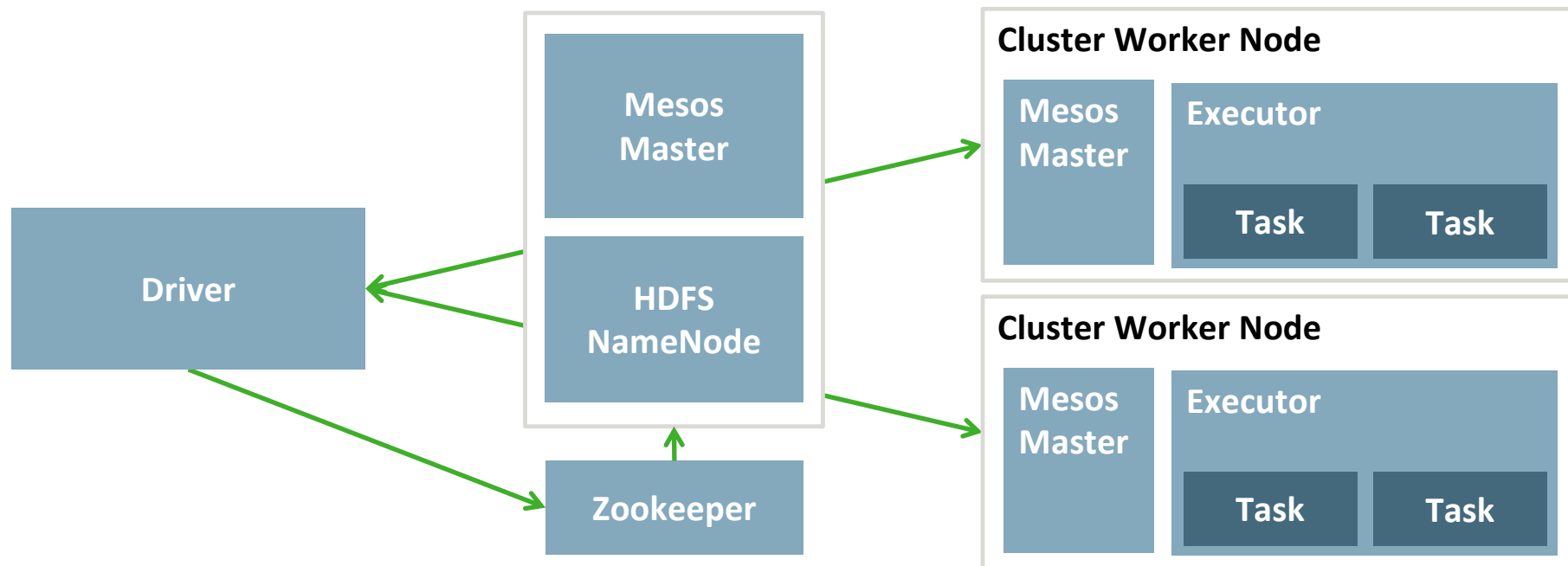
# Usabilidade do YARN

- Configuração simples:
  - Precisa ter um sistema Hadoop / YARN funcionando
  - Definir variáveis de ambiente
    - **HADOOP\_CONF\_DIR** ou **YARN\_CONF\_DIR** para o diretório de configuração do lado do client do cluster Hadoop
    - Os arquivos de configuração são usados para se conectar ao gerenciador de recursos YARN
  - Algumas propriedades específicas do YARN que você pode definir (consulte os documentos)
  - e.g. **spark.driver.cores**: cores de CPU usados no modo cluster
- Modos de deploy:
  - **yarn-cluster**: O driver do Spark é executado dentro de um processo master de um app gerenciado pelo YARN no cluster
  - **yarn-client**: O driver é executado em um processo do client – o app master é usado apenas para alocação de recursos

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

# Spark na Arquitetura Mesos

- Nós **slave do Mesos** inciam os Executors
- Tolerância a falhas na Master a partir do Zookeeper



# Usabilidade do Mesos

- Utilizando o modo de instação padrão do Mesos
- Requer que o binário do Spark esteja disponível nos nós worker
  - e.g. no HDFS
- Configuração
  - Definir variáveis de ambiente em spark-env.sh
    - `export MESOS_NATIVE_LIBRARY=<path to libmesos.so>`. ou <caminho para libmesos.dylib> no Mac OS X
    - `export SPARK_EXECUTOR_URI=<URL do binário do Spark>`
    - Defina `spark.executor.uri` como <URL do binário do Spark>
- Modos de deploy: Suporta apenas o modo client

```
$ ./bin/spark-submit --master mesos://<mesos-host>:5050 \  
--conf spark.executor.uri=<URI-of-Spark-binary> \  
--class com.es.spark.ProcessFiles \  
target/scala-2.10/testapp.jar 1G.data
```

# Modos de execução do Mesos

- O Spark pode ser executado em dois modos no Mesos
- **fine-grained (refinado)** é o modo padrão: Cada tarefa do Spark é uma tarefa separada do Mesos
  - Várias instâncias do Spark e outras estruturas compartilham máquinas
  - Os executores aumentam / diminuem o número de CPUs à medida que executam tarefas
- **coarse-grained (não refinado)**: É criada uma tarefa do Spark de longa duração para cada worker e são agendadas mini-tasks dentro dela
  - Reduz a sobrecarga de inicialização, mas reserva recursos para toda a vida do app
  - Defina **spark.mesos.coarse=true** para escolher este modo

# Qual Gerenciador Usar?

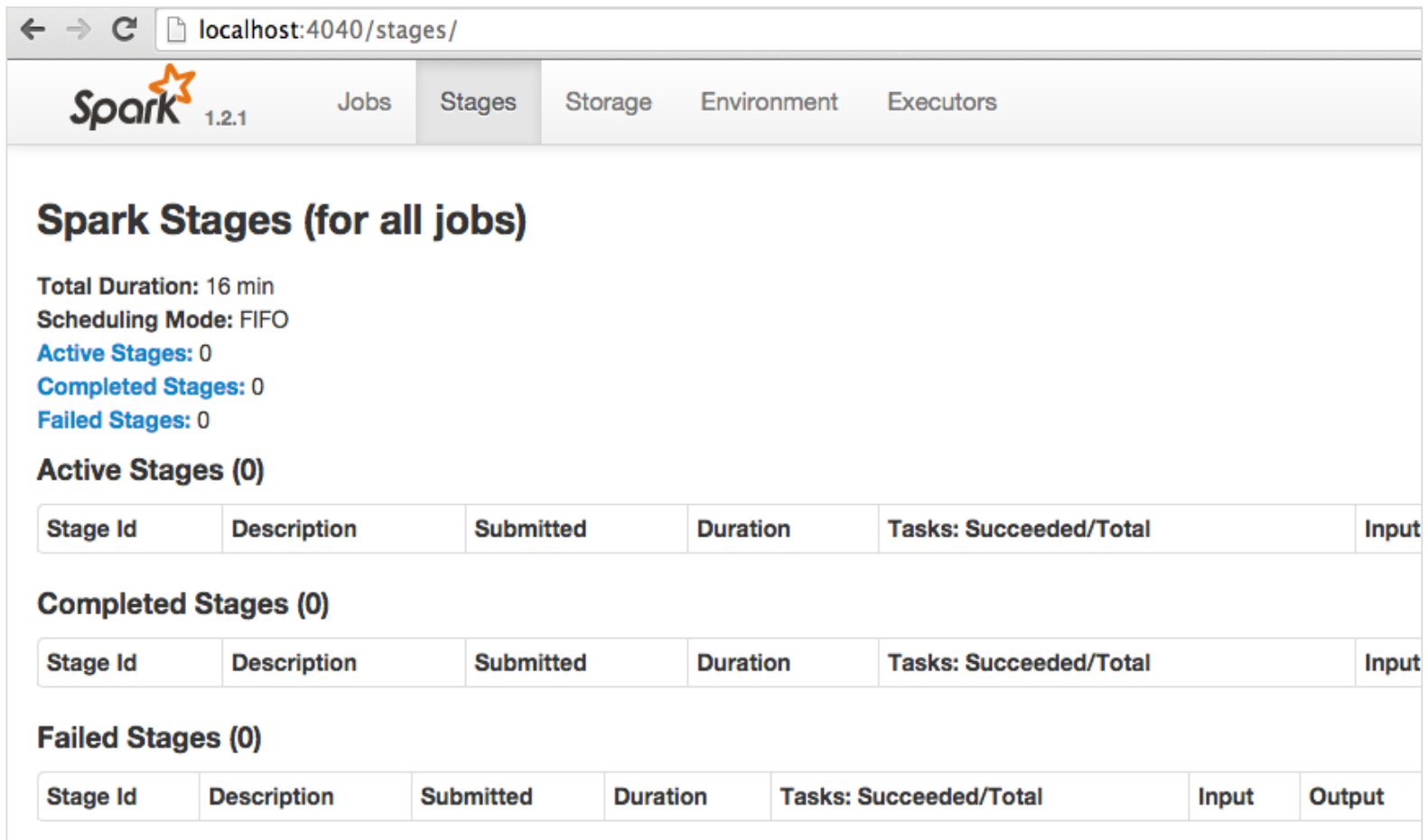
- **Execute workers do Spark dentro de nós do HDFS** com qualquer um dos gerenciadores
  - Para ter acesso rápido aos dados
- Desenvolvimento / Novo Deploys: **Standalone**
  - Simples de configurar e iniciar
  - Muito bom para execuções em Spark
- Se já estiver utilizando o Hadoop 2 : **YARN**
  - Ele estará pré-instalado
- Se a alocação refinada de recursos é importante: **Mesos**
  - Reduzirá o uso de recursos quando o job estiver menos ativo

## Parte 7.5: Logging e Debugging



# Web UI (SparkContext)

- Já vimos ela antes - é muito útil para apps independentes
  - Pode monitorar o que está acontecendo



The screenshot shows the Spark Web UI at localhost:4040/stages/. The interface includes a navigation bar with tabs for Jobs, Stages (selected), Storage, Environment, and Executors. The main content area displays 'Spark Stages (for all jobs)' with summary statistics: Total Duration: 16 min, Scheduling Mode: FIFO, Active Stages: 0, Completed Stages: 0, and Failed Stages: 0. Below these are three empty tables for Active, Completed, and Failed Stages, each with columns for Stage Id, Description, Submitted, Duration, Tasks: Succeeded/Total, and Input/Output.

← → ↻ localhost:4040/stages/

**Spark** 1.2.1 Jobs **Stages** Storage Environment Executors

### Spark Stages (for all jobs)

Total Duration: 16 min  
Scheduling Mode: FIFO  
Active Stages: 0  
Completed Stages: 0  
Failed Stages: 0

#### Active Stages (0)

Stage Id	Description	Submitted	Duration	Tasks: Succeeded/Total	Input
----------	-------------	-----------	----------	------------------------	-------

#### Completed Stages (0)

Stage Id	Description	Submitted	Duration	Tasks: Succeeded/Total	Input
----------	-------------	-----------	----------	------------------------	-------

#### Failed Stages (0)

Stage Id	Description	Submitted	Duration	Tasks: Succeeded/Total	Input	Output
----------	-------------	-----------	----------	------------------------	-------	--------


# Exibição dos Jobs

- Abaixo, nós montamos um exemplo do uso do `collect()`
  - Observe que ele está como um Job concluído
- Clicar em Job Description leva você a uma página de detalhes

Active Jobs (0)					
Job Id	Description	Submitted	Duration	Stages: Succeeded/Total	Tasks (for all stages): Succeeded/Total
Completed Jobs (1)					
Job Id	Description	Submitted	Duration	Stages: Succeeded/Total	Tasks (for all stages): Succeeded/Total
0	<a href="#">collect at &lt;console&gt;:19</a>	2015/04/17 13:49:44	0.6 s	2/2	16/16
Failed Jobs (0)					
Job Id	Description	Submitted	Duration	Stages: Succeeded/Total	Tasks (for all stages): Succeeded/Total

# Visualizando os Estágios

- Abaixo, podemos visualizar os estágios da execução do Job
  - Existem dois
  - Clicando na descrição do estágio você será levado a uma página de detalhes do estágio



JobsStagesStorageEnvironmentExecutors

Spark shell application UI

## Details for Job 0

Status: SUCCEEDED  
Completed Stages: 2

### Completed Stages (2)

Stage Id	Description		Submitted	Duration	Tasks: Succeeded/Total	Input	Output	Shuffle Read	Shuffle Write
1	<a href="#">collect at &lt;console&gt;:19</a>	<a href="#">+details</a>	2015/04/17 13:49:44	88 ms	8/8				
0	<a href="#">map at &lt;console&gt;:14</a>	<a href="#">+details</a>	2015/04/17 13:49:44	0.4 s	8/8				1012.0 B

# Detalhes do Estágio

## Details for Stage 1

Total task time across all tasks: 0.5 s

► Show additional metrics

### Summary Metrics for 8 Completed Tasks

Metric	Min	25th percentile	Median	75th percentile	Max
Duration	57 ms	58 ms	59 ms	62 ms	62 ms
GC Time	0 ms	0 ms	0 ms	0 ms	0 ms

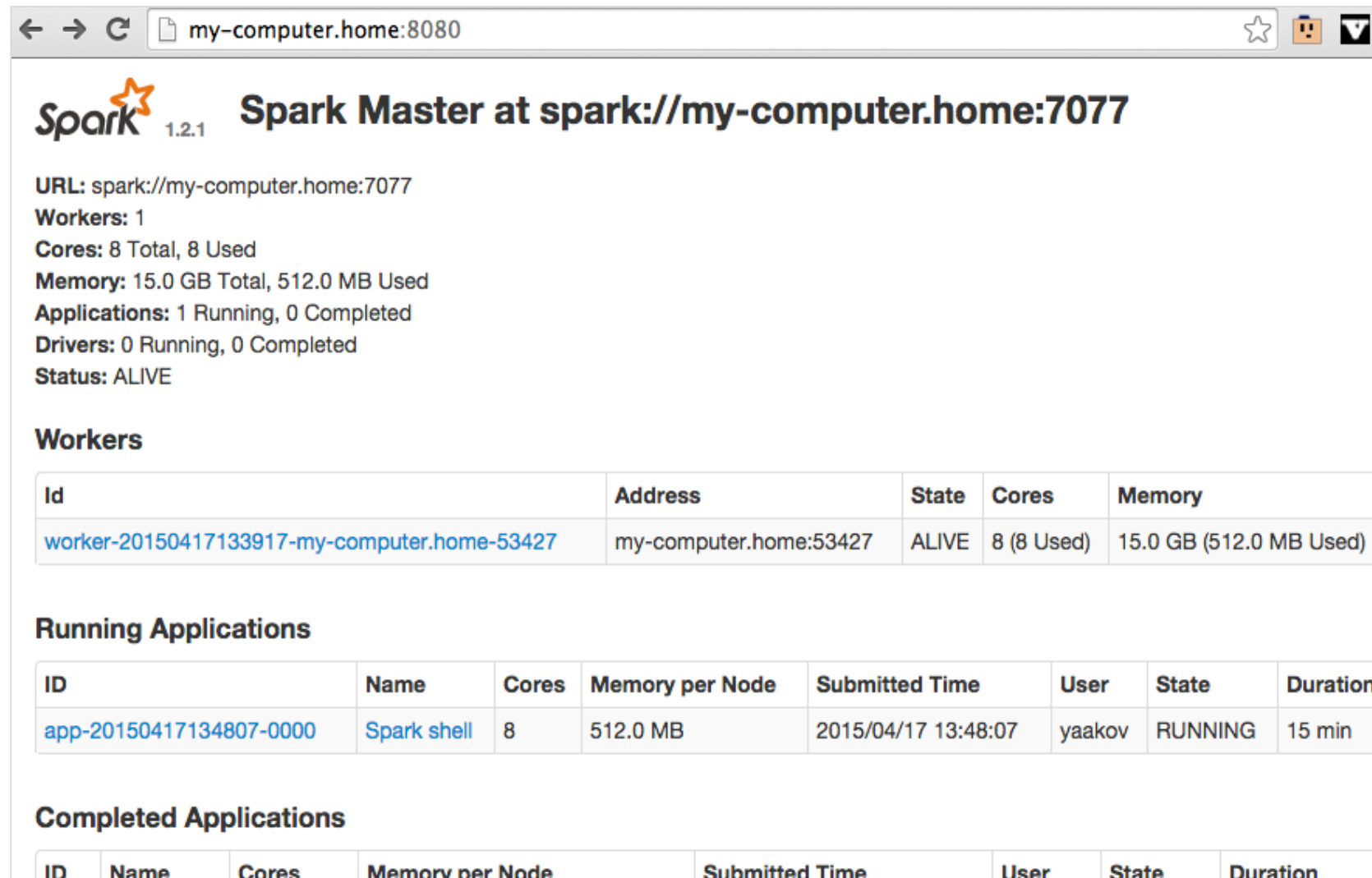
### Aggregated Metrics by Executor

Executor ID	Address	Task Time	Total Tasks	Failed Tasks	Succeeded Tasks	Input	Output	Shuffle Read	Shuffle Write	Shuffle Spill (Memory)	Shuffle Spill (Disk)
0	my-computer.home:53515	0.6 s	8	0	8	0.0 B	0.0 B	0.0 B	0.0 B	0.0 B	0.0 B

### Tasks

Index	ID	Attempt	Status	Locality Level	Executor ID / Host	Launch Time	Duration	GC Time	Errors
0	8	0	SUCCESS	PROCESS_LOCAL	0 / my-computer.home	2015/04/17 13:49:44	59 ms		
1	9	0	SUCCESS	PROCESS_LOCAL	0 / my-computer.home	2015/04/17 13:49:44	58 ms		
3	11	0	SUCCESS	PROCESS_LOCAL	0 / my-computer.home	2015/04/17 13:49:44	62 ms		

# Master UI: <master-host>:8080



The screenshot shows the Spark Master UI in a web browser. The address bar displays 'my-computer.home:8080'. The page title is 'Spark Master at spark://my-computer.home:7077'. The Spark logo and version '1.2.1' are visible. The URL is 'spark://my-computer.home:7077'. The status is 'ALIVE'. The workers section shows one worker with ID 'worker-20150417133917-my-computer.home-53427', address 'my-computer.home:53427', state 'ALIVE', 8 cores (8 used), and 15.0 GB (512.0 MB used) memory. The running applications section shows one application with ID 'app-20150417134807-0000', name 'Spark shell', 8 cores, 512.0 MB memory, submitted at '2015/04/17 13:48:07', user 'yaakov', state 'RUNNING', and duration '15 min'. The completed applications section is empty.

**Spark** 1.2.1 **Spark Master at spark://my-computer.home:7077**

URL: spark://my-computer.home:7077  
Workers: 1  
Cores: 8 Total, 8 Used  
Memory: 15.0 GB Total, 512.0 MB Used  
Applications: 1 Running, 0 Completed  
Drivers: 0 Running, 0 Completed  
Status: ALIVE

**Workers**

Id	Address	State	Cores	Memory
<a href="#">worker-20150417133917-my-computer.home-53427</a>	my-computer.home:53427	ALIVE	8 (8 Used)	15.0 GB (512.0 MB Used)

**Running Applications**

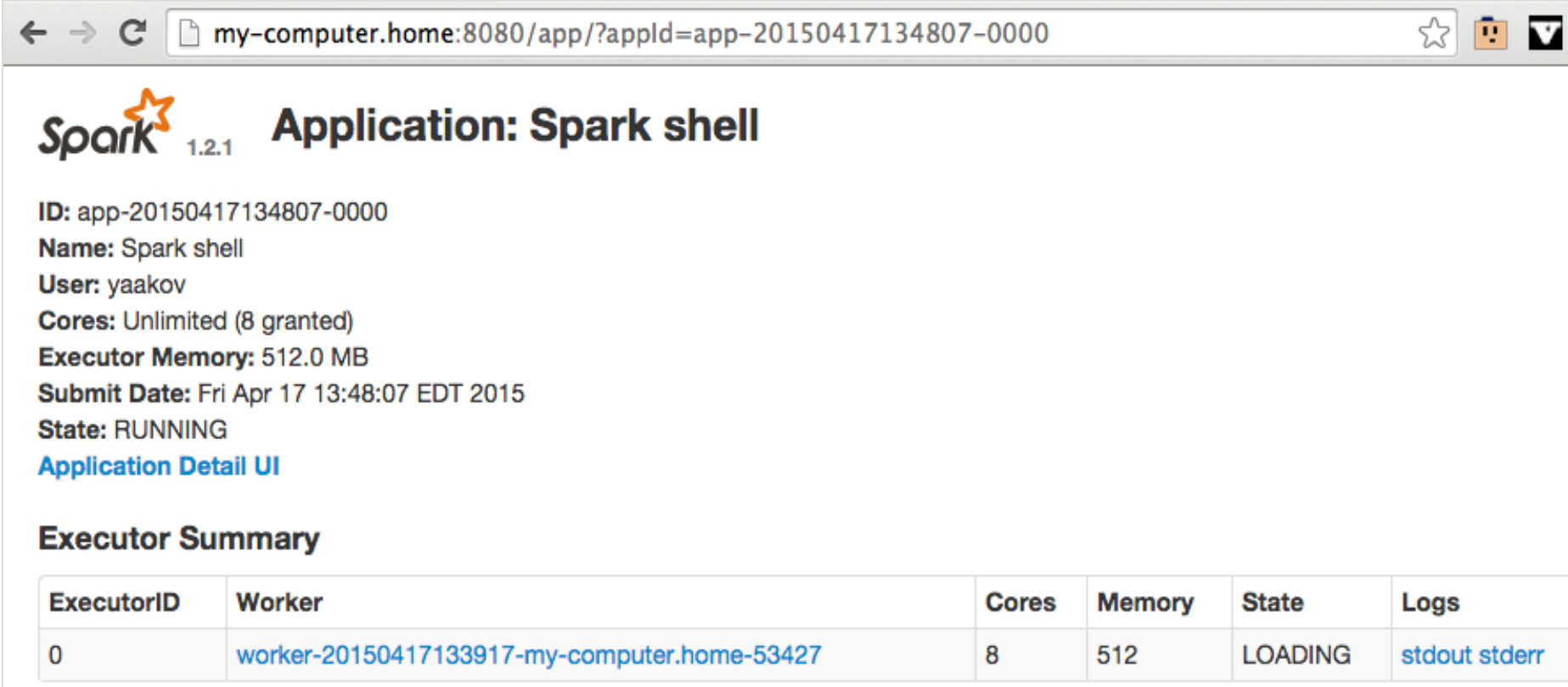
ID	Name	Cores	Memory per Node	Submitted Time	User	State	Duration
<a href="#">app-20150417134807-0000</a>	<a href="#">Spark shell</a>	8	512.0 MB	2015/04/17 13:48:07	yaakov	RUNNING	15 min

**Completed Applications**

ID	Name	Cores	Memory per Node	Submitted Time	User	State	Duration
----	------	-------	-----------------	----------------	------	-------	----------

# Master UI: Detalhes da Aplicação

- Esta é a página de detalhes para o spark shell
  - Observe como você pode acessar stdout e stderr diretamente



The screenshot shows a web browser window with the address bar displaying `my-computer.home:8080/app/?appId=app-20150417134807-0000`. The page header features the Spark logo (1.2.1) and the title "Application: Spark shell". Below this, the application details are listed: ID: app-20150417134807-0000, Name: Spark shell, User: yaakov, Cores: Unlimited (8 granted), Executor Memory: 512.0 MB, Submit Date: Fri Apr 17 13:48:07 EDT 2015, and State: RUNNING. A link "Application Detail UI" is provided. The "Executor Summary" section contains a table with the following data:

ExecutorID	Worker	Cores	Memory	State	Logs
0	<a href="#">worker-20150417133917-my-computer.home-53427</a>	8	512	LOADING	<a href="#">stdout stderr</a>

# Logging

- Os logs da Master ficam em `<spark>/logs`
  - Arquivos de logs são nomeados com o prefixo do nome de usuário de máquina de quem acessou
  - e.g. **spark-student-org.apache.spark.deploy.master.Master-1-my-computer.home.out**
- Os logs de aplicação ficam em `<spark>/work`
  - Em um subdiretório que é criado quando um app é iniciado
  - Para cada aplicativo, há um arquivo para registro de stdout e stderr
  - Os registro de logs também são visíveis na Master UI, conforme visto anteriormente
    - Mas nem todas as saídas – e.g A saída INFO não é visível lá

# Customizando o Logging

- Customize o logging criando o arquivo *conf/log4j.properties*
  - O arquivo *conf/log4j.properties.template* pode servir como base
  - Abaixo, ilustramos como alterar o nível de log root para WARN
  - Isso reduz alguns dos múltiplos registros que o Spark produz
  - Este arquivo será detectado e usado automaticamente
  - Você também pode usar a opção `--files` do `spark-submit` para enviar este arquivo junto com o jar do seu app
  - Certifique-se de distribuir o arquivo `log4j.properties` para todos os nós

```
# log4j.properties.template - INFO level to console
log4j.rootCategory=INFO, console
# Remaining detail omitted ...
```

```
# log4j.properties - WARN level to console
log4j.rootCategory=WARN, console
# Remaining detail omitted ...
```



# Perguntas de revisão

- Como você escreve uma aplicação Spark?
- Como você executa uma aplicação Spark?
- O que é um gerenciador de cluster e quais são compatíveis com o Spark?

# Resumo

- Existem dois tipos principais usados para escrever um aplicativo Spark
  - **SparkSession**: Gerenciar operações de cluster
  - **SparkSession.Builder**: Factory para configurar o SparkSession
- As aplicações Spark geralmente são executadas usando o comando spark-submit
- Os gerenciadores de cluster gerenciam o processamento distribuído do cluster.
  - Existem três gerenciadores de cluster suportados
  - **Standalone**: Vem com Spark e é uma solução exclusiva dele
  - **YARN**: Gerenciador de cluster Hadoop 2
  - **Mesos**: Gerenciador de cluster criado em Berkeley
    - Pode fornecer alocação dinâmica de recursos para uso eficiente de recursos

## Parte 8: Visão geral sobre Spark Streaming

- Introdução ao Streaming
- Spark Streaming (1.0+)
- [Opcional] Aprofundamento em Spark Streaming (1.0+)
- Spark Structured Streaming (2.0+)
- Consumindo dados do Kafka

# Parte 8.1: Introdução ao Streaming

# A Evolução do Big Data

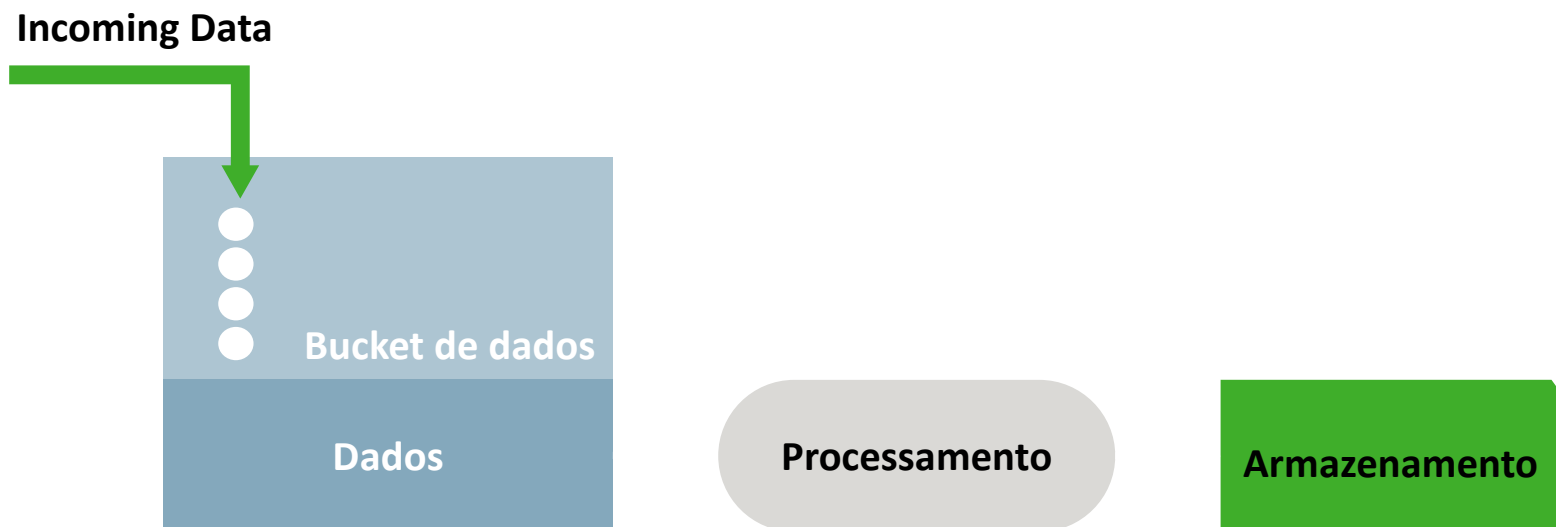
- V1: Antigamente
  - **Tomada de decisão:** Orientada a lotes (horas/dias)
  - **Caso de uso:** Relatórios
- Atualmente: Todos acima **mais** a necessidade de processar **streams (fluxos) de dados**
  - **Tomada de decisão:** (Perto de ser) em tempo real (milisegundos, segundos)
  - **Casos de uso:** Alertas (médicos/segurança, detecção de fraude...)
    - Dispositivos conectados / Internet of Things (IoT)
    - E muito mais
  - Necessidade de processamentos/análises mais rápidas

# Visão geral de Streaming de Dados

- **Streaming de dados** são gerados continuamente
  - Frequentemente por muitas fontes de dados (e.g. IoT)
  - Geralmente com payloads muito menores
- As necessidades de processamento de dados em streaming incluem:
  - Processamento **sequencial** e **incremental**
  - Processamento de **registro em registro** de forma **contínua**
  - **Transformações** em dados, incluindo agregações
    - e.g. filtros, médias, contagens
- O processamento em streaming é um requisito fundamental
  - Muitas tecnologias suportam isso (e.g. Storm, Flink, etc.)
  - Vamos focar nas habilidades do Spark

# Arquitetura em Alto Nível do Streaming

- **Bucket de dados:** Captura / armazena dados de entrada
  - Opções: **Kafka**, MQ, Amazon Kinesis
- **Processamento:** Processamento com baixa latência
  - Opções: **Spark**, Storm, Flink, ...
- **Armazenamento:** Armazenamento dos dados — geralmente em um NoSQL
  - NoSQL: HBase, Cassandra ..



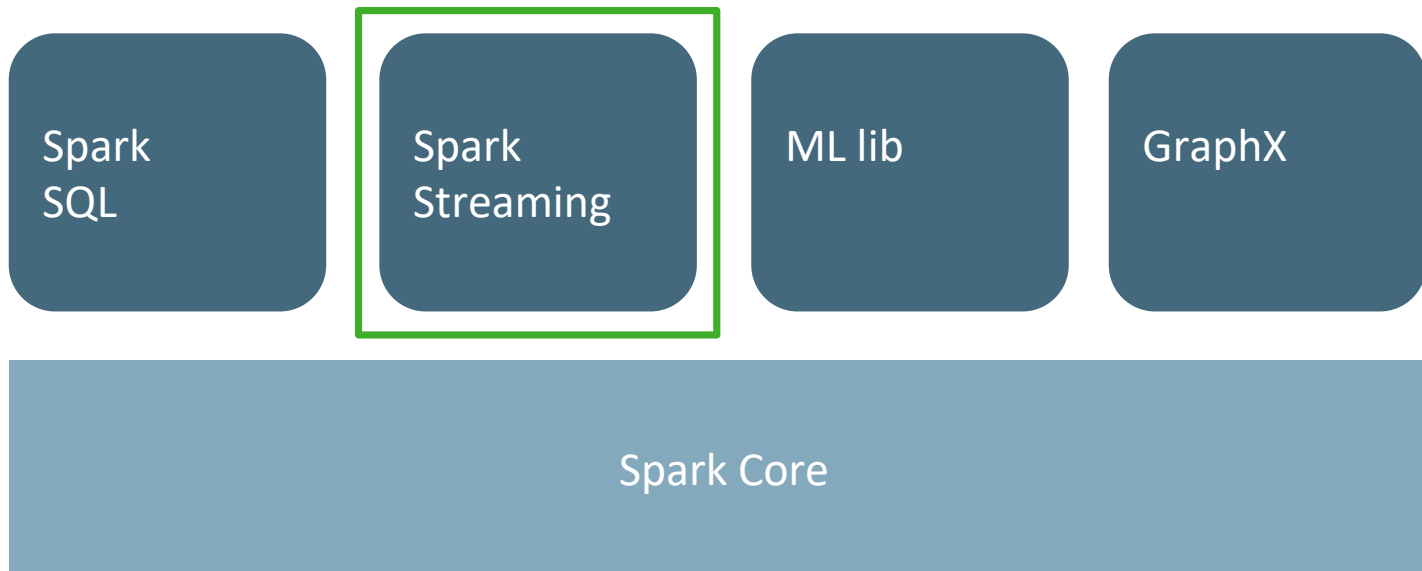
# Spark Streaming — Duas opções

- **Spark Streaming:** Lançada nas primeiras versões (0.7+)
  - Baseado em RDD, API complexa
  - Novos esforços indo sendo gastos no streaming estruturado
  - Continuará a ser suportado
- **Spark Streaming Estruturado:** Lançado no Spark 2.1
  - Baseado em DataFrame, arquitetura melhorada
  - API simples (parte disso devido ao uso do DataFrame)
- Ambos se beneficiam dos recursos do Spark
  - Pode se integrar com transformações, gráficos, ML
  - Alta performance, escalável, tolerância a falhas



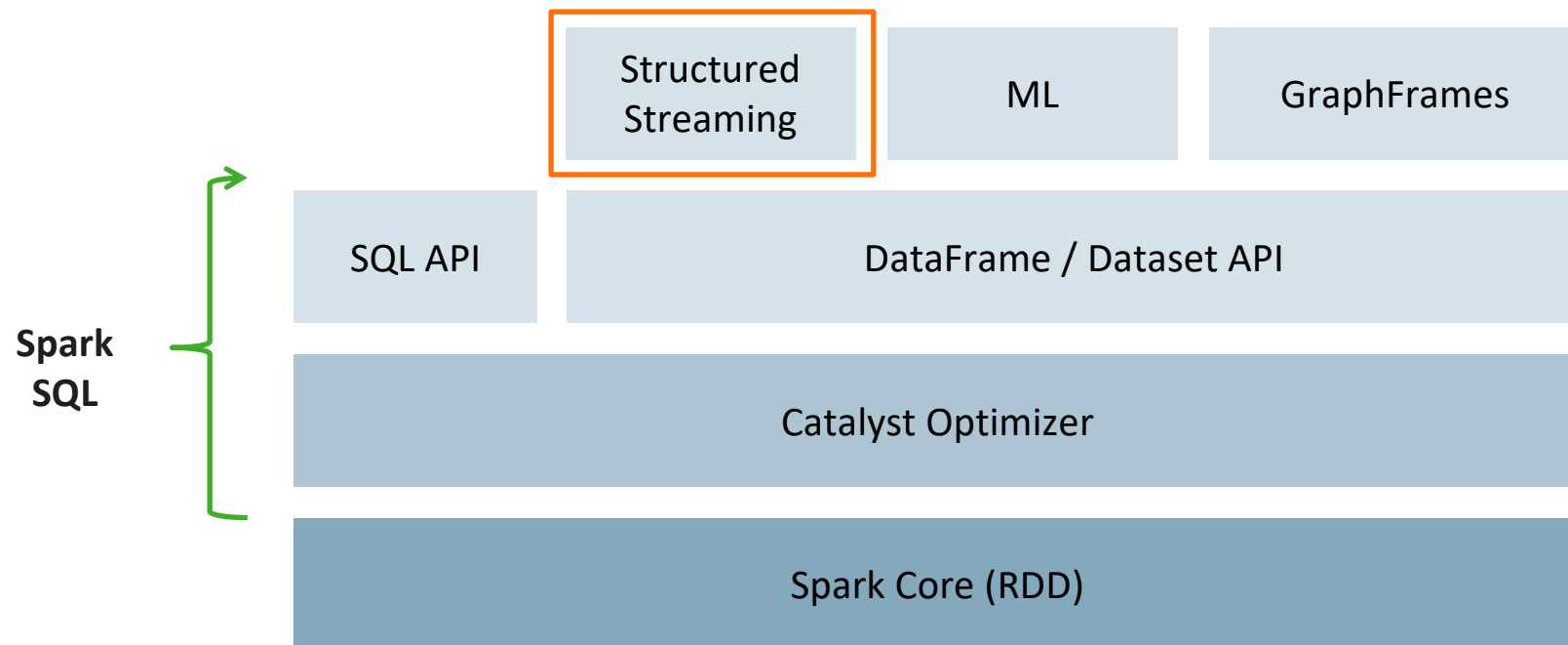
# Lugar do Spark Streaming no Ecosystema Spark

- Construído com base no **Spark Core**
  - Baseado em RDD



# Lugar do Spark Streaming Estruturado no Ecossistema Spark

- Construído com base no Spark SQL
  - Baseado em DataFrame
  - Herda os benefícios do Spark SQL



# Comparando os Recursos dos Sistemas de Streaming

Feature	Storm	Spark Streaming	Spark Structured Streaming	Flink
Modelo de Processamento	Por padrão, baseado em eventos	Micro lotes	Micro lotes	Baseado em eventos + Micro lotes
Operações Windowing	Suportado pelo Trident	Sim	Sim	Sim
Latencia	Milisegundos	Segundos		Milisegundos
Processamento interno	pelo menos uma vez	exatamente uma vez	exatamente uma vez	exatamente uma vez
Queries iterativas	NÃO	SIM	SIM	NÃO
Join com dados estáticos	NÃO	SIM	SIM	NÃO

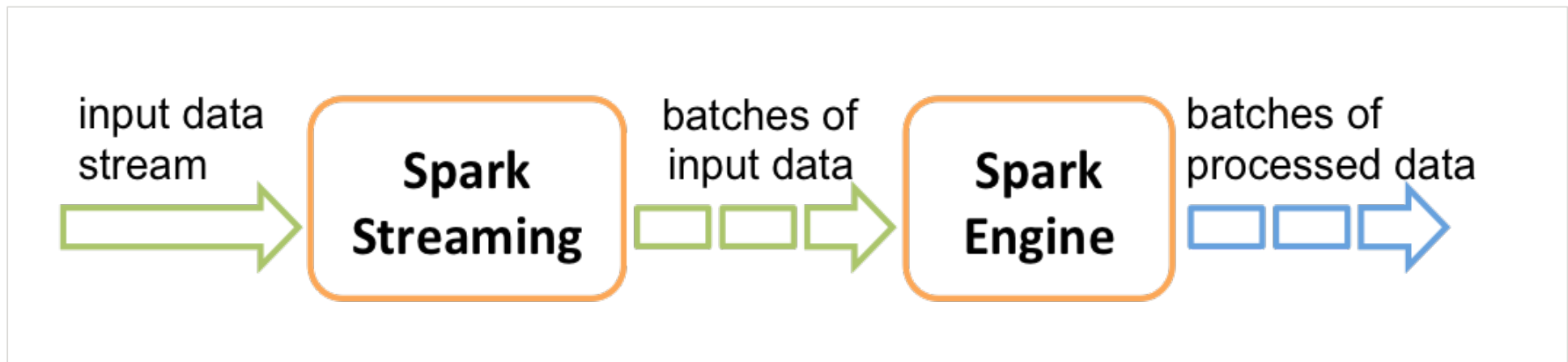
## Parte 4.2: Spark Streaming

# Conceitos chave

- Transforma dados de streaming brutos em dados processados
  - Baixa latência e tolerância a falhas
- **DStream**: Sequencia de RDDs que representam a entrada
- **Transformações**: Modificar um RDD DStream para outro RDD
  - Fornece transformações Stateless e Stateful
- **Operação de resultado**: Envia os dados para uma fonte externa
  - Salva em algum armazenamento
  - Processa o lote de alguma outra maneira

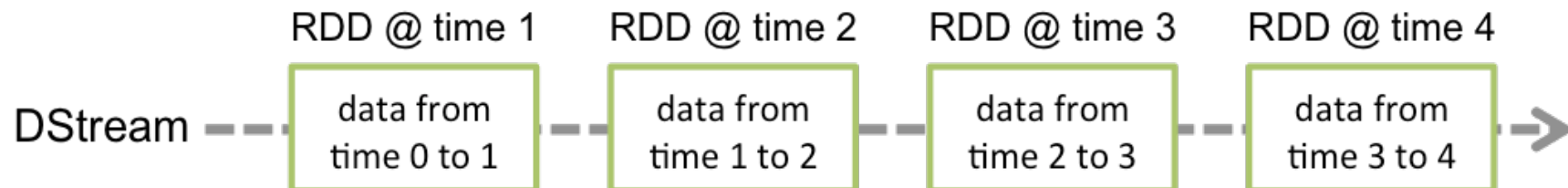
# Como isso funciona?

- Estrutura o processamento Streaming como uma série de **pequenos, stateless, processos específicos em lotes** (DStreams)
  - Divide a transferência atual em lotes de um intervalo (segundos)
  - Cada lote se torna um RDD, processado por meio de operações RDD
  - Os resultados processados são retornados em lotes

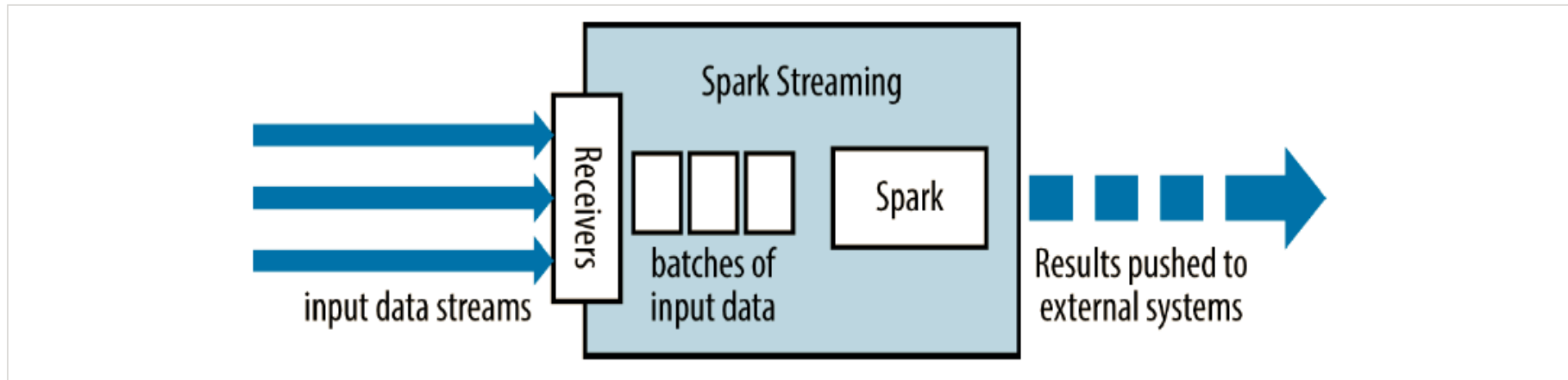


# Discretized Steams (DStreams)

- **DStream**: Sequencia de RDDs que representam dados
  - Os dados chegam com o tempo
    - A partir de muitas origens — Flume, Kafka, HDFS ...
  - Os dados são divididos em microlotes (RDDs), e então são processados
- Tipos de operações do DStream:
  - **Transformação**: Modificar o dado (resultando em outro DStream)
    - Suporta as operações padrão de RDD
    - Fornece operações stateful— e.g. operações windowed
  - **Resultado**: Envia para uma fonte externa



# Arquitetura – Receptores



- Cada DStream de entrada é relacionado a um **receptor**
  - Receptores são tasks executando nos executors das aplicações.
  - Eles coletam os dados e transformam em RDDs
  - Receptores são “tasks de longa duração”
- Os dados de Streaming podem ser recebidos pela rede
  - Kafka, Flume, sockets, etc.
- Também pode criar um fluxo carregando dados periodicamente de um armazenamento externo (e.g. HDFS)



# Arquitetura — Lotes



- Os dados Dstream de fontes de entrada são agregados em lotes
  - Com base no **intervalo de lote** (configurável)
  - Os dados dentro de um determinado intervalo de tempo são adicionados ao lote
    - O intervalo geralmente é de 0,5 seg. a muitos segundos (com base em suas necessidades)
  - O lote é fechado no final do intervalo
  - Cada lote se torna um RDD
- O processamento é distribuído entre máquinas, como RDDs normais
  - O streaming pode continuar conforme o processamento em lote ocorre (feito em paralelo)
  - Pode se integrar com RDDs não DStream (por exemplo, por meio de uma junção)

# Processamento com DStream

- Cada lote de entrada é convertido para um RDD
  - Conforme o intervalo do lote passa, novos RDDs são gerados continuamente
  - Contendo os dados capturados nesse intervalo
- DStreams podem ser transformadas
- Portanto, um DStream gera RDDs periodicamente por qualquer:
  - **Empacotando dados** em um RDD inicial
  - **Transformando um RDD** gerado por um DStream pai

# Tipos de Transformações do DStream

- **Stateless:** A transformação em um lote não depende do lote anterior
  - Transformações de RDD comuns, como `map()`, `filter()`, `countByValue()`, `reduceByKey()`, etc.
  - e.g. filtrar uma determinada palavra / string de um Streaming de texto
- **Stateful:** Usa dados / resultados de lotes anteriores para processar o lote atual
  - Baseado em **continuidade** e rastreabilidade de estado ao longo do tempo
  - As operações incluem **window()** e **countByValueAndWindow()**
  - Exemplos de casos de uso: Calcule a% de aumento de preço por negociação para uma determinada ação na NASDAQ

# Exemplo Simples — Visão geral

- Vamos exemplificar com um programa simples que
  - Configura um receptor de streaming que lê de um socket
    - Utiliza lotes com duração de 5 segundos
    - Filtra todas as linhas de entrada, exceto aquelas que contêm a string "Scala"
  - Grava a entrada processada no console

- 

Nossos dados de entrada são criados por meio do programa **nc** (netcat)

- Digitar o texto de entrada no console do nc para enviar a informação pela rede
- Configure o host e a porta no nc para os clients se conectarem

# Exemplo de Código (1 de 3 — Inicialização)

- Importar os objetos necessários
- Definir o **batchDuration**: Intervalo de tempo para processar novos dados
- Criar o **StreamingContext**: Ponto de entrada principal para streaming
  - Fornecer métodos para criar DStreams de várias fontes de entrada

```
import org.apache.spark.streaming.StreamingContext
import org.apache.spark.streaming.dstream.DStream
import org.apache.spark.streaming.Duration
import org.apache.spark.streaming.Seconds

// Trecho de código mostrando apenas código de Streaming
//spark.sparkContext.getConf
val conf = new
SparkConf().setMaster("local[2]").setAppName("StreamingExample")
val batchDuration = Seconds(5)
val ssc = new StreamingContext(conf, batchDuration)
// Agora você está pronto para iniciar o processamento
```

## Exemplo de Código (2 de 3 — Configurar a entrada)

- Configurar a fonte de entrada
  - Aqui, recebemos o payload a partir da rede (um socket)
- Criar um DStream para a fonte de entrada
  - `socketTextStream()` cria um DStream para leitura de payload de rede
- Transformar usando RDDs
  - Neste caso, filtrar por linhas que contém o texto "Scala"
  - E faça o print delas

```
val port = 9999 // Utilize a porta apropriada para o seu app
val lines : ReceiverInputDStream[String] =
    ssc.socketTextStream("localhost", port)
val scalaLines = lines.filter(_.contains("Scala"))
scalaLines.print()
```

## Exemplo de Código (3 de 3 — Processamento)

- Três métodos do `StreamingContext` controlam o streaming:
  - **`start()`**: Inicia o processamento (deve ser feito apenas uma vez)
    - Como o exemplo abaixo
  - **`stop()`**: Parar o processamento de forma manual
  - **`awaitTermination()`**: Aguardar o processamento ser finalizado
    - Como o exemplo abaixo
- Execute isso como qualquer outro app Spark
  - e.g. usando `spark-submit`, ou o shell

```
ssc.start() // Inicialização  
ssc.awaitTermination() // Execute até o processo terminar
```

# Programa Completo

```
package com.mycompany.streaming
import org.apache.spark._
import org.apache.spark.streaming._

object StreamingExample {
  def main(args: Array[String]) {
    // Criando o Contexto do Streaming
    val ssc = new StreamingContext("local[2]",
                                   "StreamingExample", Seconds(5))
    // Criando o DStream da origem (socket)
    val lines = ssc.socketTextStream("localhost", 9999)
    // Filtrando (cria um novo DStream)
    val scalaLines = lines.filter(_.contains("Scala"))
    scalaLines.print() // Resultado

    ssc.start()
    ssc.awaitTermination()
  }
}
```



# Resultados

- À esquerda, emulamos uma fonte de rede com o netcat
  - Nós digitamos um pouco, esperamos, digitamos um pouco mais
- À direita está o output da execução do nosso programa
  - Você pode ver que a saída veio em vários lotes(nosso intervalo de lotes foi de 5 segundos = 5,000 ms)

```
$ nc -lk 9999
```

```
Scala 1
```

```
English 2
```

```
Java 3
```

```
Burp 4
```

```
Scala 5
```

```
Buzz
```

```
6
```

```
French 7
```

```
Scala 8
```

```
[info] Running  
com.mycompany.streaming.StreamingExample
```

```
-----  
Time: 1429590630000 ms  
-----
```

```
-----  
Time: 1429590640000 ms  
-----
```

```
Scala 1
```

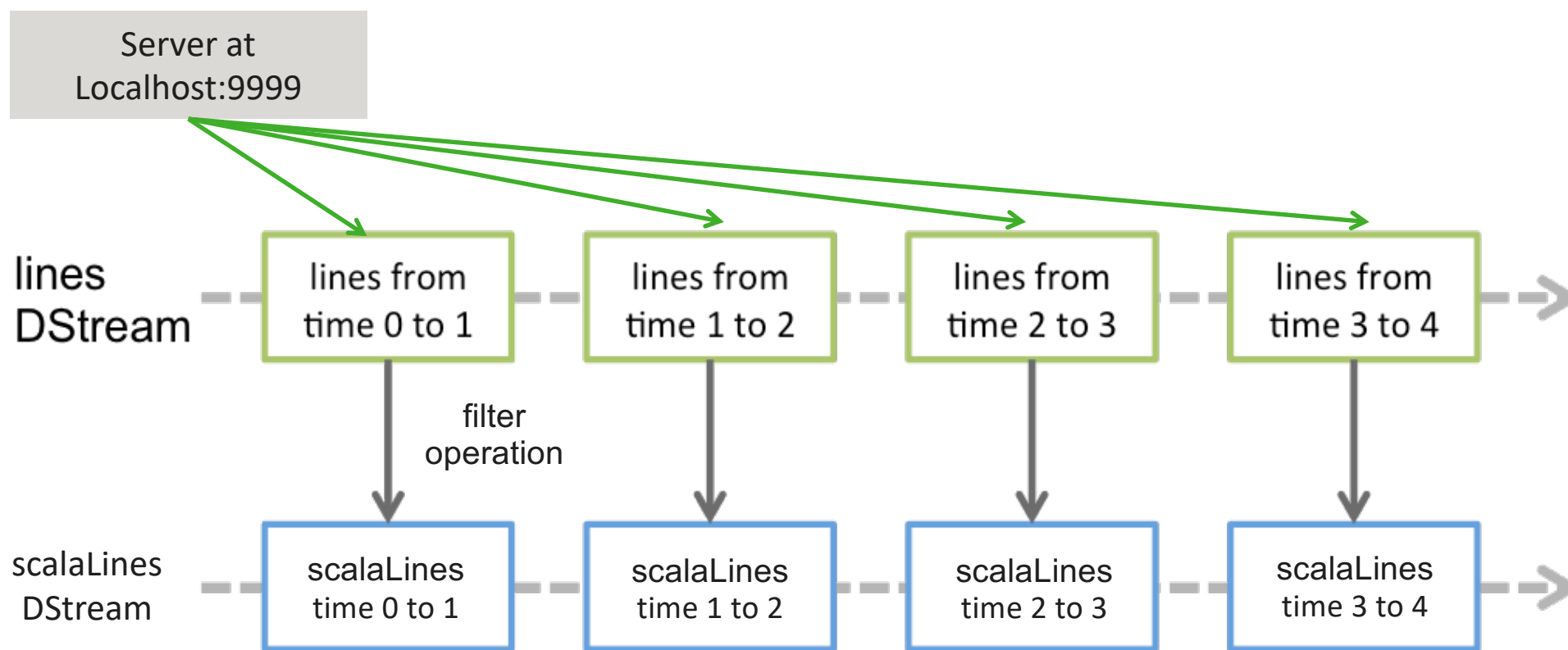
```
Scala 5
```

```
-----  
Time: 1429590650000 ms  
-----
```

```
Scala 8
```

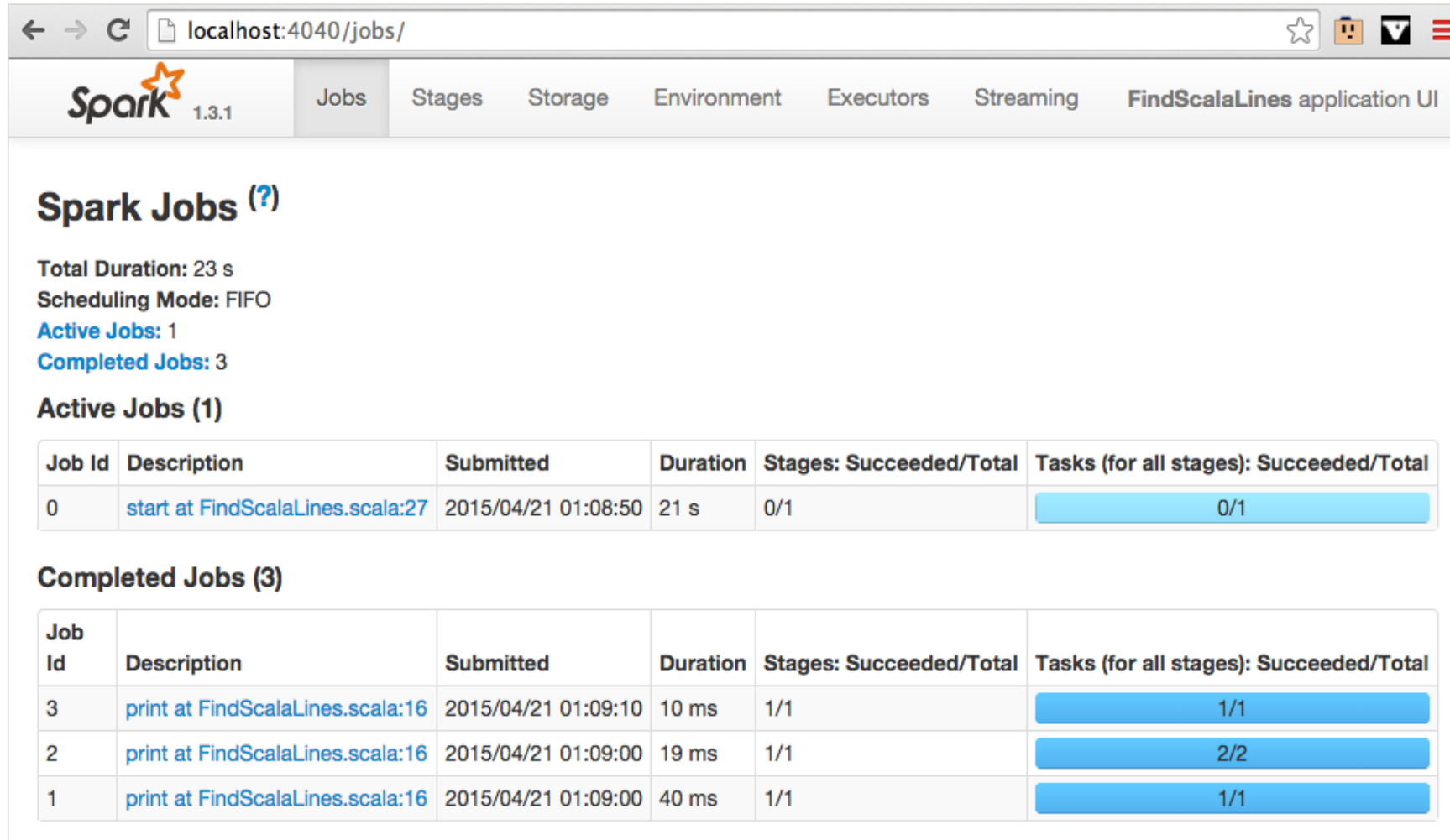
# DStreams

- Abaixo, ilustramos como a entrada da rede é agrupada em DStreams e depois transformada



# Driver UI

- Você pode ver que vários Jobs foram executados



The screenshot shows the Spark Driver UI at localhost:4040/jobs/. The interface includes a navigation bar with tabs for Jobs, Stages, Storage, Environment, Executors, Streaming, and FindScalaLines application UI. The main content area displays 'Spark Jobs (?)' with summary statistics: Total Duration: 23 s, Scheduling Mode: FIFO, Active Jobs: 1, and Completed Jobs: 3. Below this, there are two sections: 'Active Jobs (1)' and 'Completed Jobs (3)'. Each section contains a table with job details.

**Active Jobs (1)**

Job Id	Description	Submitted	Duration	Stages: Succeeded/Total	Tasks (for all stages): Succeeded/Total
0	<a href="#">start at FindScalaLines.scala:27</a>	2015/04/21 01:08:50	21 s	0/1	0/1

**Completed Jobs (3)**

Job Id	Description	Submitted	Duration	Stages: Succeeded/Total	Tasks (for all stages): Succeeded/Total
3	<a href="#">print at FindScalaLines.scala:16</a>	2015/04/21 01:09:10	10 ms	1/1	1/1
2	<a href="#">print at FindScalaLines.scala:16</a>	2015/04/21 01:09:00	19 ms	1/1	2/2
1	<a href="#">print at FindScalaLines.scala:16</a>	2015/04/21 01:09:00	40 ms	1/1	1/1

# MINI-LAB: Reveja a Documentação

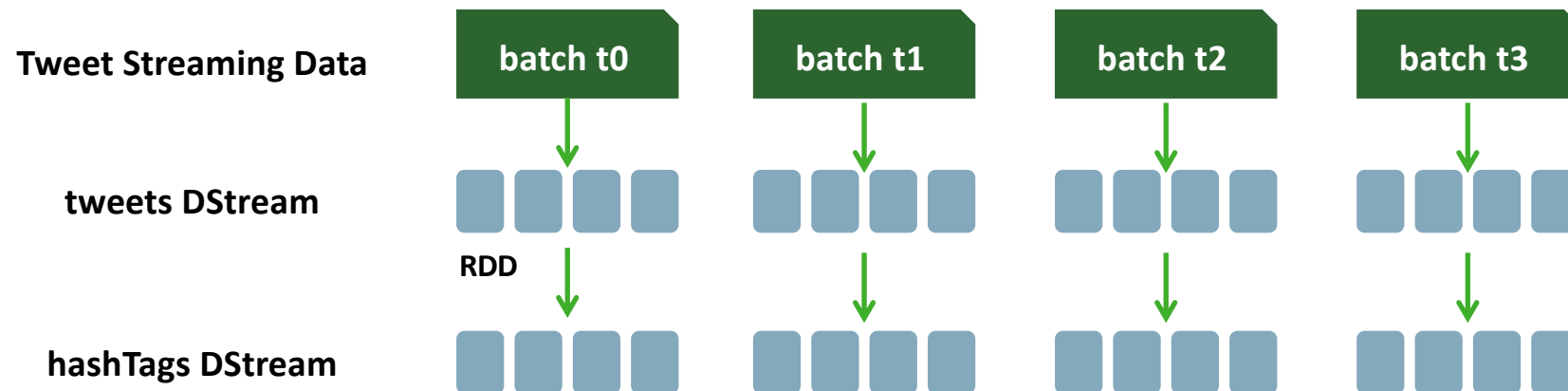
- Acesse os documentos do Spark <http://spark.apache.org/docs/latest/>
  - Na barra de pesquisa no canto superior esquerdo, busque pelos objetos abaixo
  - Reveja suas documentações
  - **StreamingContext** (`org.apache.spark.streaming`)
    - Reveja os métodos `socketTextStream()`, `start()`, `stop()`, e `awaitTermination()`
    - Se quiser, reveja outros métodos
  - **DStream** e **ReceiverInputDStream** (`org.apache.spark.streaming.dstream`)
    - Observe os diferentes tipos de transformação, e.g. `filter()`

# Lab 8.1: Spark Streaming (Vamos fazer juntos)

## Parte 8.4: [Opcional] Aprofundamento em Spark Streaming

# Transformação Stateless

- Abaixo, ilustramos as transformações em Streams do Twitter
  - Suponha que tenha um Streaming transmitindo tweets do Twitter para o Spark
  - Para cada lote, um RDD é gerado no DStream
    - Ilustrado pelo DStream **tweets** abaixo
  - Em seguida, mapeamos/filtramos o DStream para obter as Tags de cada tweet
    - Ilustrado pelo DStream **hashTags** abaixo
  - Linhagem direta de um RDD pai para um RDD transformado



# Transformação Stateless para DStream

- Suporta muitas transformações RDD normais, incluindo:
  - Lembre-se — Estas são aplicadas para cada RDD no stream

Transformation	Description	Example	f's
map(f)	Aplica f para cada elemento do DStream	<code>ds.map(x =&gt; x*2)</code>	<code>f: T -&gt; U</code>
flatMap(f)	Semelhante ao mapa, mas pode gerar mais de um resultado por elemento	<code>ds.flatMap( x =&gt; x.split (" "))</code>	<code>f: T -&gt; Iterable[U]</code>
filter(f)	Filtrar por cada elemento quando f for verdadeiro	<code>ds.filter( x=&gt; x % 2 == 1)</code>	<code>f: T -&gt; Boolean</code>
repartition(n)	Change number of partitions	<code>ds.repartition(10)</code>	NA (numerical)
reduceByKey(f)	Alterar o número de partições	<code>ds.reduceByKey( (x,y) =&gt; x+y)</code>	<code>f: (T, T) -&gt; T</code>
groupByKey()	Valores de grupo com a mesma chave (para dados de pares)	<code>ds.groupByKey()</code>	NA
mapPartitions(func)	Como o mapa, mas é executado em toda a partição, não em cada elemento		



# Exemplo de Transformação Stateless

- Abaixo, mapeamos um DStream que representa um log de acesso
  - O método `map()` produz um par de DStream  
(IP address, 1)
  - O método `reduceByKey()` gera um RDD com inputs no formato abaixo  
(basicamente visitas por IP)  
(IP address, total access count)
- Essas transformações são muito semelhantes às RDD normais
  - A diferença? À medida que os dados entram, novos RDDs são criados a cada intervalo de lote para conter os dados

```
// Considere que accessLogsDStream é criado em algum lugar
val ipDStream = accessLogsDStream.map(entry =>
    (entry.getIpAddress(), 1) )

val ipCountDStream = ipDStream.reduceByKey( (x,y) => x+y)
```

# Dados de Múltiplos DStreams

- Transformações Stateless também podem combinar dados de vários DStreams em cada intervalo de tempo
  - DStreams têm as mesmas transformações relacionadas à Joins que RDDs
- As operações de exemplo são semelhantes às suas contrapartes RDD, e.g.
  - `cogroup()`, `join()`, and `leftOuterJoin()`
- Mesclar dois streams usando o operador `union()` do
  - Semelhante a RDDs regulares
  - Use `StreamingContext.union()` para múltiplos streams

# Transformação Stateless Avançada

- No fundo, juntamos dois DStreams que são produzidos por transformações de map e reduce
  - O map transforma um stream de dados de IP em um par DStream (IP address, content size)
  - reduceByKey processa o total de bytes para um IP
  - Por último, juntamos a contagem de solicitações por IP (mostrado anteriormente) com o total de bytes para o IP fornecido

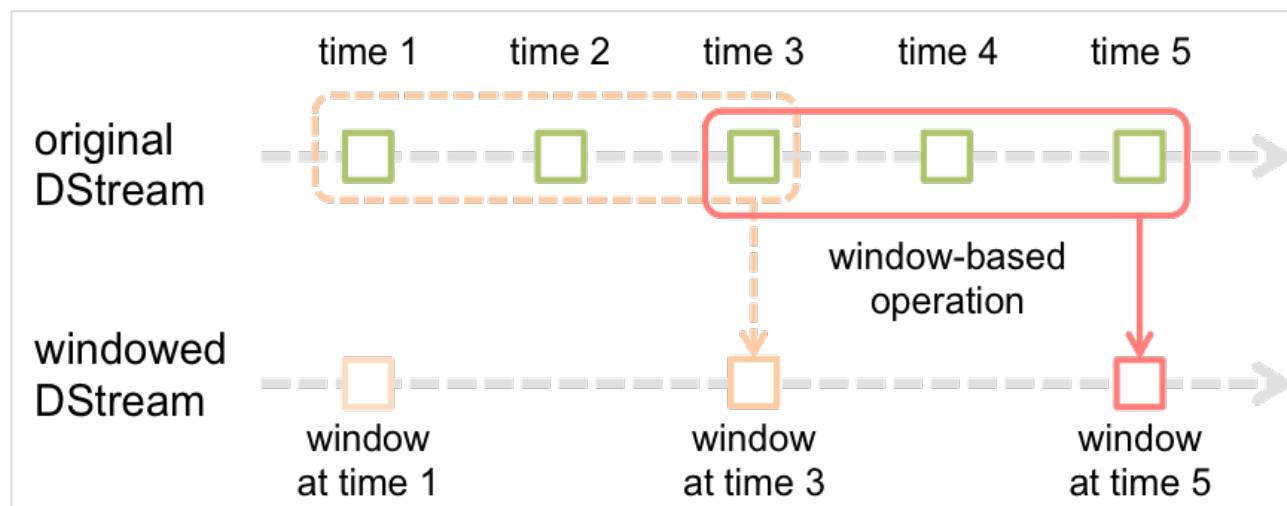
```
// Considere que accessLogsDStream é criado em algum lugar
val ipBytesDStream = accessLogsDStream.map(entry =>
    (entry.getIpAddress(), entry.getContentSize()))

val ipBytesSumDStream =
    ipBytesDStream.reduceByKey((x, y) => x + y)

val ipBytesRequestCountDStream =
    ipCountsDStream.join(ipBytesSumDStream)
```

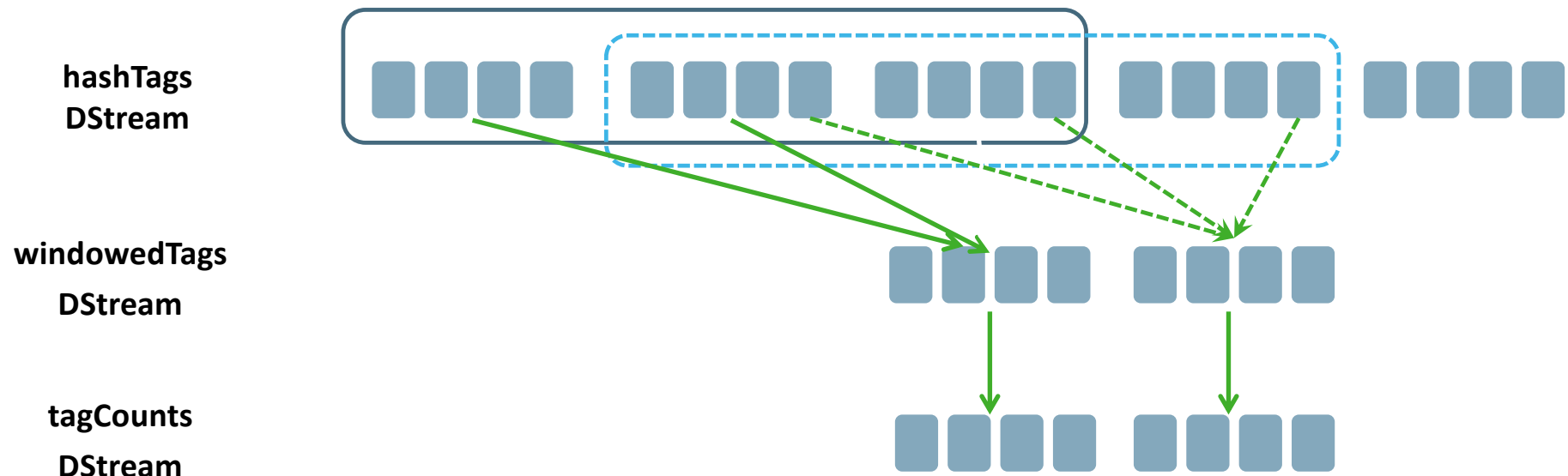
# Transformações Stateful (Windowed)

- Estas rastreiam os dados ao longo do tempo
  - Dados anteriores são usados para gerar ou executar as transformações atuais
- Cada operação Stateful, necessita de 2 parâmetros
  - Ambos são múltiplos do intervalo de lote
  - **Window Duration (“Janela de duração”)**: Quando lotes de dados anteriores usar
  - **Slide Duration (frequência dos resultados)**: Com que frequência o novo DStream processa os resultados
  - Abaixo, temos window duration = 3, e slide duration = 2



# Ilustrando a Transformação Stateful (ou Windowed)

- Abaixo, transformamos o DStream `hashTags` DStream de forma Stateful
  - Com window duration = 3, e slide duration = 1
  - O resultado vai para o DStream **windowedTags**
  - Podemos processar isso ainda mais - por exemplo, contando a ocorrência de cada tag na janela, conforme mostrado no DStream **tagCounts**



# API de Transformações Stateful

- Processamentos Stateful (ou Windowed) transformam os dados utilizando uma “Janela dinâmica de dados”
  - Definida pelos parâmetros de tamanho de “janela”(“window”) e intervalo dinâmico (“sliding”)
- O uso mais simples é apenas pegar uma janela de dados

```
def window(windowDuration: Duration,  
           slideDuration: Duration)
```

  - Ambos os argumentos devem ser múltiplos do intervalo do lote
- Vamos demonstrar isso por meio de um programa Stateful de contagem de palavras
  - Ele produz contagens de palavras de dados em uma “janela dinâmica”
  - Vamos dar uma olhada no código a seguir - veremos as versões Stateless e Stateful

# Exemplo de contagem de palavras sem o uso de “janelas”

- O exemplo abaixo deve parecer familiar - é apenas a contagem de palavras comum que já conhecemos
  - Mas ainda há uma diferença — ele obtém os dados de um stream e continua em execução até ser interrompido
  - Os dados são loteados

```
val ssc =  
  new StreamingContext("local[2]", "WordCount", Seconds(5))  
  
val lines = ssc.socketTextStream("localhost", 9999)  
val words = lines.flatMap(_.split(" "))  
val pairs = words.map(word => (word, 1))  
// Sem o uso de “janelas”  
val wordCounts = pairs.reduceByKey(_ + _)  
wordCounts.print()  
ssc.start()  
ssc.awaitTermination()
```

# Exemplo de contagem de palavras com o uso de “janelas”

- No exemplo abaixo, `pairsWindow` é uma versão com “janela” dos pares de um RDD
  - Com Window Duration=15, Slide Duration=3
- `wordCountsWindow` agora contém dados de uma “janela dinâmica”

```
val ssc =  
  new StreamingContext("local[2]", "WordCount", Seconds(3))  
  val lines = ssc.socketTextStream("localhost", 9999)  
  val words = lines.flatMap(_.split(" "))  
  val pairs = words.map(word => (word, 1))  
  // With windowing  
  // Uso simples de “janela”.  
  val pairsWindow = pairs.window(Seconds(15), Seconds(3))  
  val wordCountsWindow = pairsWindow.reduceByKey(_ + _)  
  wordCountsWindow.print()
```



# Resultados da contagem de palavras com “Janela”

- Abaixo, podemos ver as entradas
- À direita, o resultado
  - Observe como as palavras persistem durante um intervalo de janela
  - e.g. o "a" aparece nos primeiros 3 conjuntos de resultados (já que a duração da nossa janela é 3)

```
$ nc -lk 9999
a a a b b c
d d d e e f
g g g h h i
```

```
(b,2)
(a,3)
(c,1)
-----
```

```
(d,3)
(b,2)
(f,1)
(e,2)
(a,3)
(c,1)
-----
```

```
(d,3)
(b,2)
(f,1)
(e,2)
(a,3)
(c,1)
-----
```

```
(d,3)
(h,2)
(f,1)
(e,2)
(i,1)
(g,3)
-----
```

```
(h,2)
(i,1)
(g,3)
-----
```

```
(h,2)
(i,1)
(g,3)
-----
```

# reduceByKeyAndWindow()

- Permite que você especifique uma “janela” para o reduce.  
`reduceByKeyAndWindow`(reduceFunc: (V, V) ⇒ V, windowDuration: Duration, slideDuration: Duration): DStream[(K, V)]
  - Combina o uso de janelas com reduce
  - Você obtém o mesmo resultado da versão anteriores
- No entanto — ambas as versões têm que somar janelas completas de dados para cada intervalo de duração da janela

```
val ssc =  
  new StreamingContext("local[2]", "WordCount", Seconds(5))  
  val lines = ssc.socketTextStream("localhost", 9999)  
  val words = lines.flatMap(_.split(" "))  
  val pairs = words.map(word => (word, 1))  
  // Com janelas mais complexas  
  val wordCountsWindow =  
    pairs.reduceByKeyAndWindow((a:Int,b:Int) => (a + b),  
      Seconds(15), Seconds(5))  
  wordCountsWindow.print()
```

# Outras operações de “Janela”

- **countByWindow()**: Retorna a quantidade de elementos em cada janela
- **countByValueAndWindow()**: Retorna contagens para cada valor
  - Todos esses métodos retornam um DStream, com cada RDD contendo o valor apropriado para seu lote pai
- Abaixo, temos alguns exemplos

```
val ipDStream =  
  accessLogsDStream.map{entry => entry.getIpAddress()}  
val ipAddressRequestCount =  
  ipDStream.countByValueAndWindow(Seconds(30), Seconds(10))  
val requestCount =  
  accessLogsDStream.countByWindow(Seconds(30), Seconds(10))
```

# Operações de resultados

- **print(num: Int)**: Exibe os primeiros “num” elementos de cada lote
- **print()**: Exibe os primeiros 10 elementos de cada lote
  - DStream é materializado para realizar isso
- **save()**: Salva elementos de um DStream em um diretório separado.
  - `ipAddressRequestCount.saveAsTextFiles("outputDir", "txt")`
- **saveAsHadoopFiles()**: Salva cada RDD como arquivo Hadoop
  - É possível passar parâmetros de prefixo e sufixo
  - Existem também os métodos `saveAsTextFiles()`, `saveAsObjectFiles()`
- **foreachRDD(foreachFunc)**: Aplicar uma função em cada RDD

```
ipAddressRequestCount.foreachRDD { rdd =>
  rdd.foreachPartition { partition =>
    val connection = openConnection
    partition.foreach { record => connection.send (record))
    connection.close()
  }
}
```

# Fontes de entrada

- Fontes principais:
  - Stream of Files
  - Akka Actor Stream
- Fontes populares:
  - Sockets
  - Apache Kafka
  - HDFS
  - Apache Flume
  - Twitter
  - Para incluir esses receptores adicionais, adicione o artefato Maven **spark-streaming-[projectname]\_2.10**,
  - e.g. **spark-streaming-kafka\_2.10**

# Interface Interna do DStream (1 de 2)

- Define como gerar o lote em cada intervalo
  - Lista de DStreams dependentes (pais)
    - `def dependencies: List[DStream[_]]`
  - Duração do intervalo: Espaço de tempo em que processa RDDs
    - `def slideDuration: Duration`
  - Método para processar o RDD em um determinado momento
    - `def compute(validTime: Time): Option[RDD[T]]`
- Exemplo: Dstream mapeada
  - Dependencias: **Um DStream pai**
  - Duração do intervalo: **O mesmo do DStream pai**
  - Método de processamento: **Aplicar uma função map no DStream's pai**

# Interface Interna do DStream (2 de 2)

- Exemplo: DStream em Janela
  - Dependencias: **Um DStream pai**
  - Duração do intervalo: **Duração do intervalo da janela**
  - Método de processamento: **Aplicar o método union em todos os RDDs do DStream pai na janela atual**
- Example: Receptor de entradas DStream (pela rede)
  - Dependencies: **Nenhuma**
  - Slide duration: **Duração do lote**
  - Compute method: **Cria um RDD com todos os dados recebidos no último intervalo de lote**

# Tolerância a Falhas

- Para tolerância a falhas, os dados de entrada podem ser replicados
  - Replicado em dois nós - tolera falha de um único worker
    - Padrão para streams que recebem dados pela rede (Kafka, Flume, ...)
  - Apenas os dados brutos de entrada são replicados na memória
    - RDDs não transformados
- O gerenciador de memória Sparks, denominado Block Memory, mantém os dados replicados enquanto for necessário
  - E os RDDs se lembram de sua linhagem
- Os dados perdidos devido à falha do worker são reprocessados usando os dados de entrada brutos e a linhagem
  - Portanto, os dados transformados também são tolerantes a falhas
- O **checkpointing** é muito crítico para transformações Stateful



# Checkpointing

- Salva o estado do RDD periodicamente em um sistema de arquivos confiável
  - HDFS ou S3
- **Porque?** RDDs DStream Stateful podem ter uma linhagem muito grande
  - Os RDDs de resultado dependem de RDDs de lotes anteriores, a cadeia de dependência continua crescendo com o tempo
  - Portanto, o tempo de recuperação pode ser grande para dados acumulados ao longo de um longo período
  - Tamanhos de tasks / tempo de lançamento também aumentam
- **Solução:** Faça checkpoints periodicamente
  - Para se recuperar da falha, só precisa voltar ao último checkpoint
  - Normalmente, o checkpoint = 5-10 vezes o intervalo de janela do DStream

# Operações 24/7

- O Spark Streaming pode ser executado no modo 24/7, mesmo se um worker ou driver falharem.
- Para que o streaming funcione 24/7, é necessário um sistema de armazenamento confiável, como S3 ou HDFS para Checkpointing  
`ssc.checkpoint("hdfs://...")`
- A falta de checkpoints gera um aviso / erro mesmo na configuração local.

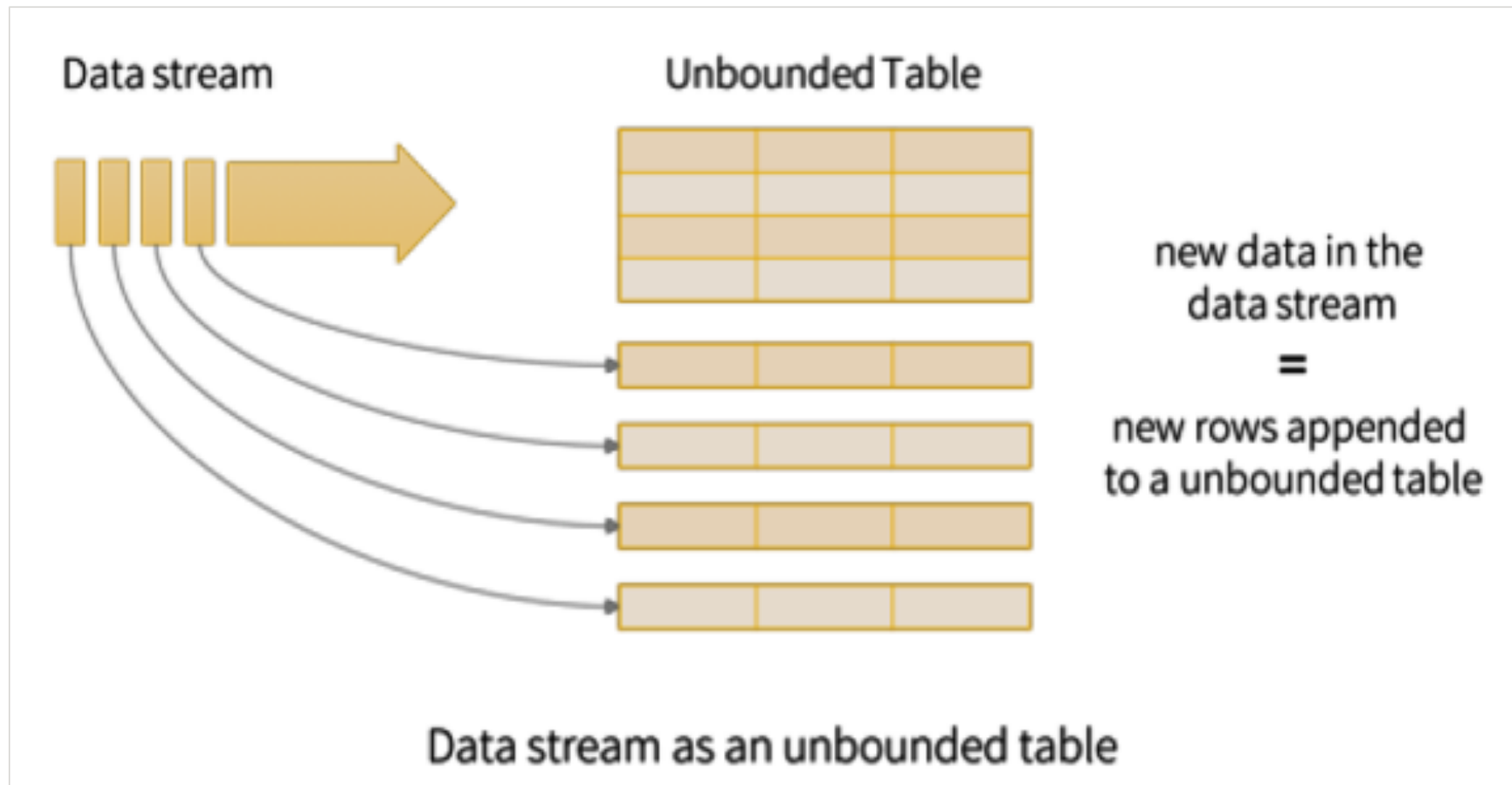
## Parte 8.5: Spark Structured Streaming (+2.0)

# Conceitos Chave

- Projetado para suportar **aplicações contínuas**:
  - Aplicação ponta a ponta que reage aos dados em tempo real
- Construído com base em DataFrames - nível superior ao Spark Streaming
  - A API de streaming é igual à API em lote (batch)!
- Novos recursos importantes para oferecer suporte a aplicativos contínuos:
  - **Jobs de streaming consistentes com jobs em lote**:
    - Escrito usando DataFrame API
  - **Integração transacional com sistemas de armazenamento**:
    - Processar dados exatamente uma vez
    - Atualiza coletores de output de forma transacional
  - **Integra-se com o resto do Spark**
    - Spark SQL, ML, etc.
    - Meta: Cada biblioteca no Spark é executada de forma incremental em Streaming Estruturado

# Como isso funciona?

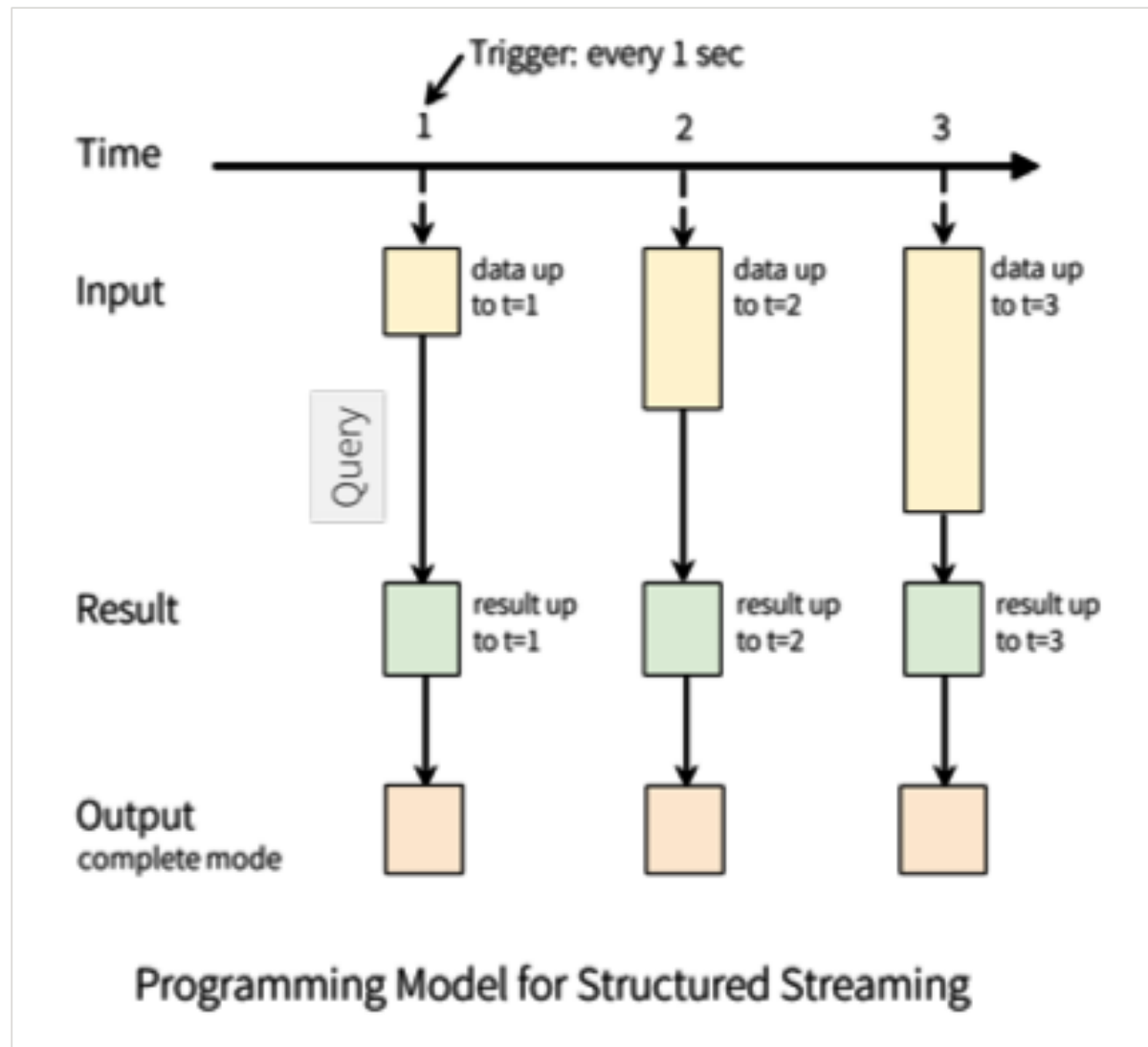
- Considere a o dado entrada do stream como uma tabela
  - Os novos dados que chegam são como uma nova linha anexada à tabela



# Tabela de Resultados

- Uma consulta na entrada cria uma tabela de resultados
  - Para cada intervalo de triggers (por exemplo, 1 segundo), novas linhas são anexadas à tabela de entrada
  - Eventualmente, a tabela de resultados é atualizada
  - Novos dados são então processados por suas transformações de consulta
- Suporta três **modos de saída**:
  - **completo**: Toda a tabela atualizada é a saída
  - **acrescentado**: Novas linhas anexadas desde que o último trigger foi executado
  - **atualizado**: Linhas atualizadas desde que o último trigger foi executado

# Tabela de Resultados



# Etapas para Streaming Estruturado

- Configure o DataFrame de entrada
  - Use **SparkSession.readStream()** para criar um **DataStreamReader**
  - Defina a fonte de entrada via **format()**
    - Atualmente, as fontes de arquivo, Kafka ou socket são suportadas
  - Defina as opções de fonte de entrada (depende do tipo de fonte)
- **Execute a consulta** para iniciar o streaming
  - Use **DataSet.writeStream()** para criar um **DataStreamWriter**
  - Defina um **intervalo de trigger** (com que frequência os dados são obtidos)
    - Padrão — o mais rápido possível depois que os dados estiverem disponíveis
  - Defina os **detalhes do coletor de output** (formato dos dados, localização, etc.)
  - Defina o **modo de output**
  - **Inicie** o processo de consulta



# Código de Exemplo – Visão Geral

- Este programa simples faz o mesmo processamento que nosso exemplo anterior de Streaming (1.x)
  - Configura um stream de entrada, com uma fonte que lê de um socket
    - Usando um intervalo de trigger de 5 segundos.
  - Filtra todas as linhas de entrada, exceto aquelas que contêm a string "Scala"
    - Usando operações padrão de DataFrame
  - Grava a entrada filtrada no console
  - Usando um stream de output
- Os dados de entrada são criados por meio do programa **nc** (netcat)

# Código de Exemplo (1 de 2 — Inicialização)

- Obtenha um **DataStreamReader** a partir da sessão via **readStream()**
  - Utilize o método **format()** para definir o formato do dado de entrada
    - Socket nesse caso
  - Utilize o método **option()** para definir qualquer opção da fonte de dados
    - Host e porta nesse caso
  - Utilize o método **load()** para carregar a entrada (de modo lazy)
    - Nenhum trabalho feito até você consumir os dados de streaming com um coletor
  - Filtre os dados — método **DataFrame.filter()** padrão

```
val lines = spark.readStream
    .format("socket")
    .option("host", "localhost")
    .option("port", 9999)
    .load()
val scalaLines = lines.filter('value.contains("Scala"))
```

## Código de Exemplo (2 de 2 — Consumir os Dados)

- Crie um **DataStreamWriter** via **writeStream()**
  - Defina o intervalo de trigger em 5 segundos
  - Defina o modo de saída como **append**
  - Defina o formato como **console**
  - Inicie o processamento a partir do método **start()**
    - Retorna uma instância de **StreamingQuery**

```
import org.apache.spark.sql.streaming.ProcessingTime

val query = scalaLines.writeStream
  .trigger(ProcessingTime("5 seconds"))
  .outputMode("append")
  .format("console")
  .start()

query.awaitTermination()
```

# Fontes e coletores suportados

- **DataStreamReader** atualmente suporta essas fontes de entrada
  - **Socket streams:** (Somente teste) lê a entrada a partir de um socket
    - Via `format("socket")`
  - **File streams:** CSV, JSON, text, Parquet
    - Via `csv()`, `json()`, `parquet()`, `textFile()`
  - **Kafka**
    - Via `format("kafka")`
- **DataStreamWriter** atualmente suporta esses coletores de saída
  - **Console:** (Para debugging) — saída para o console
    - Via `format("console")`
  - **File:** CSV, JSON, text, Parquet
    - Via `format("XXX")` onde XXX pode ser "parquet", "json", etc.
  - **Memory:** (Para debugging) — armazena a saída em uma tabela em memória
    - Via `format("memory")`
  - **foreach:** Execute processamentos arbitrários nos registros de saída
    - Via `foreach(...)`

# Resumo

- O Spark Structured Streaming é melhor do que o Spark Streaming
  - Baseado em Spark SQL / DataFrames
  - Recebe todos os benefícios — Catalyst, Tungsten, API de alto nível