Spark API/Applications

Core API

Building and Running Applications

Application Lifecycle

Logging & Debugging

Licensed for personal use only for Fernando K <fernando_kruse@dell.com> from Machine Learning at Dell Brazil (QE) @ 2019-03-12

Lesson Objectives

- Cover the Spark API more formally
- Learn to develop Spark applications

Licensed for personal use only for Fernando K <fernando_kruse@dell.com> from Machine Learning at Dell Brazil (QE) @ 2019-03-12

Spark Applications

- So far, we have been using Spark Shell
  - Stand-alone
  - Or connecting to cluster
- Shell is great for
  - Ad-hoc/interactive
  - Developing apps/Debugging
- Once code is ready, you generally code an application
  - Fairly simple using some boilerplate code
  - Can be in Scala, Python, or Java
  - We'll cover the Scala API, but all are similar

Licensed for personal use only for Fernando K <fernando_kruse@dell.com> from Machine Learning at Dell Brazil (QE) @ 2019-03-12

Application Lifecycle

Core API

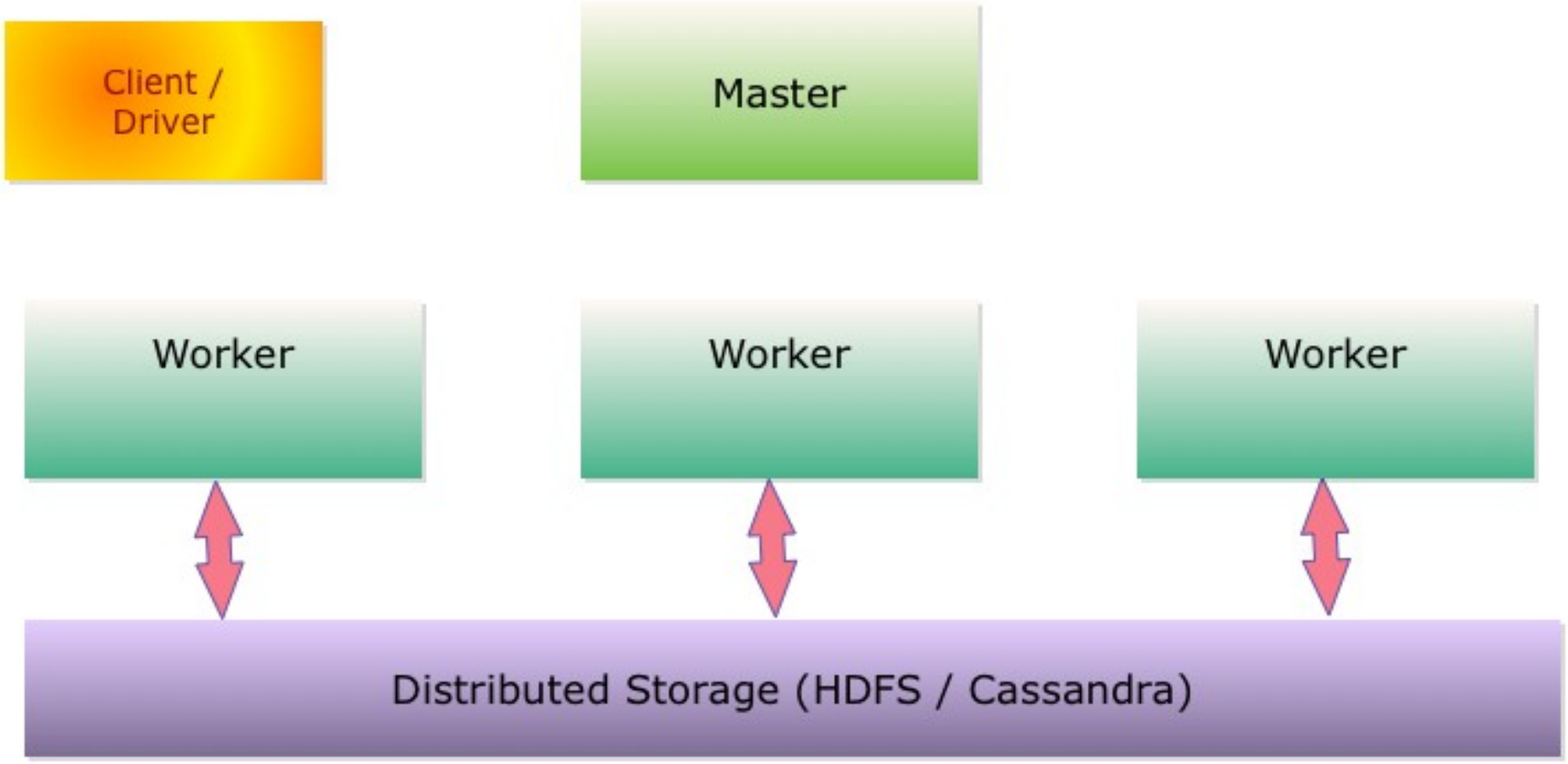
Building and Running Applications

➔ **Application Lifecycle**

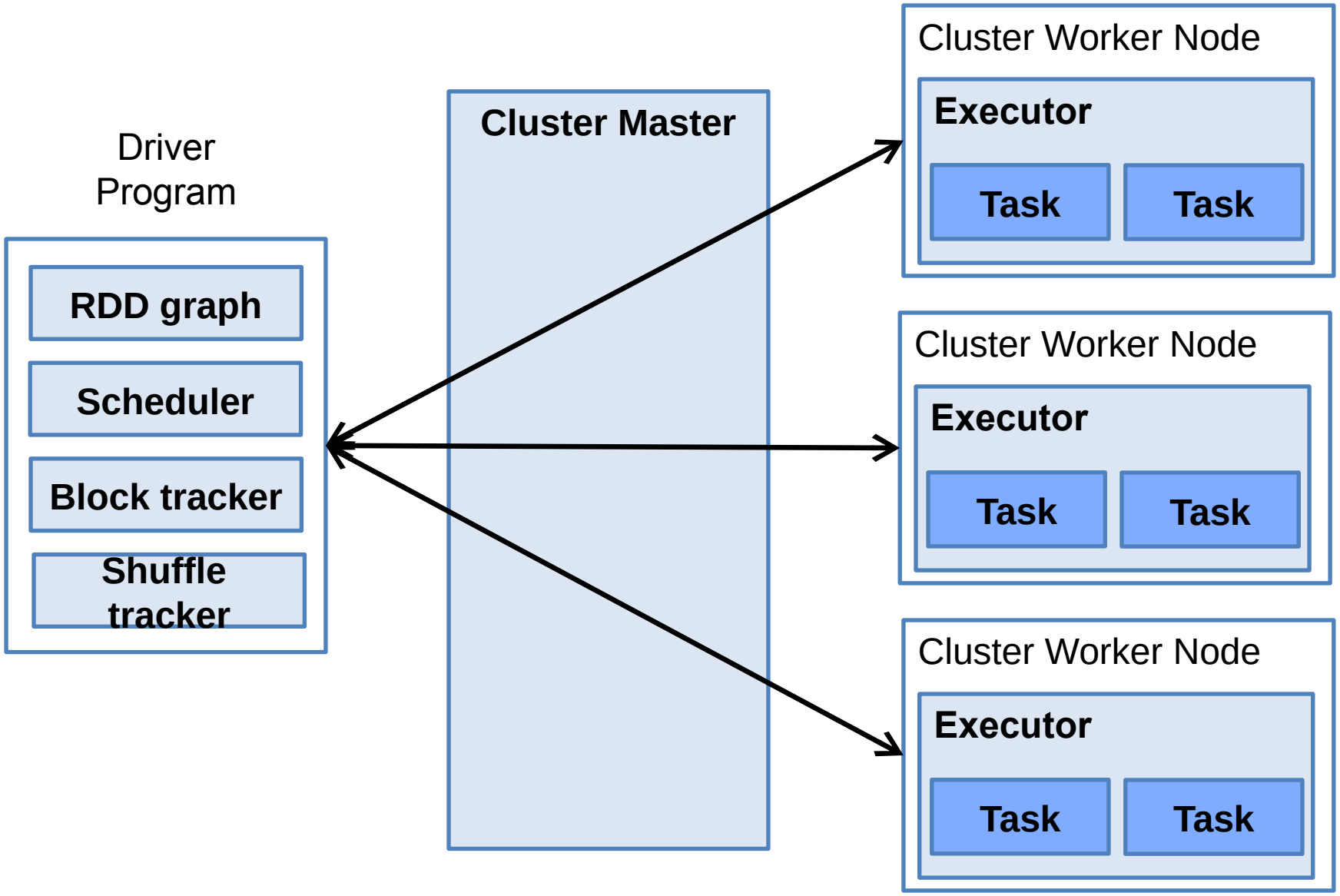
Logging & Debugging

Licensed for personal use only for Fernando K <fernando_kruse@dell.com> from Machine Learning at Dell Brazil (QE) @ 2019-03-12

Spark Runtime Components



Spark Application Architecture



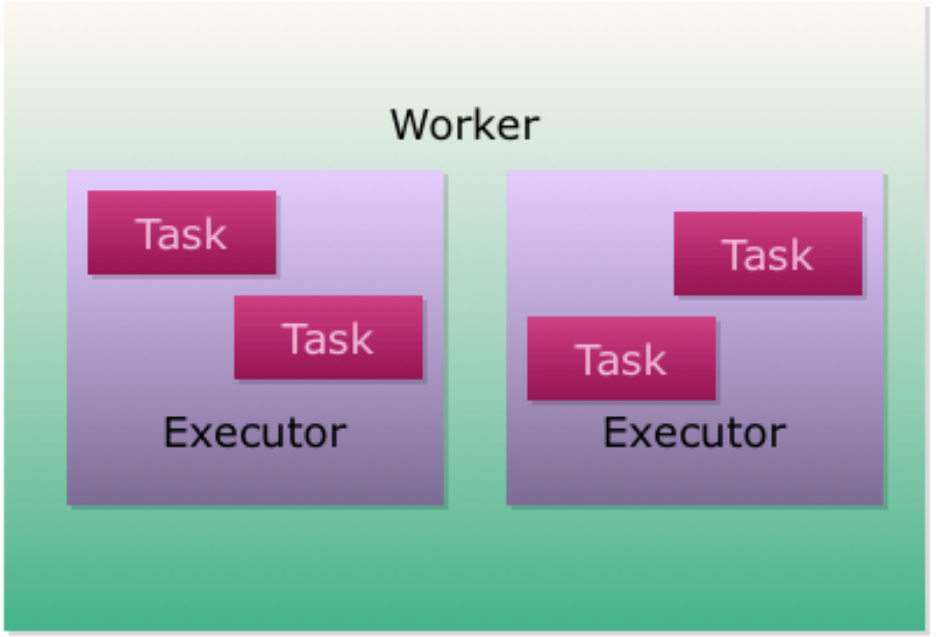
Application Driver

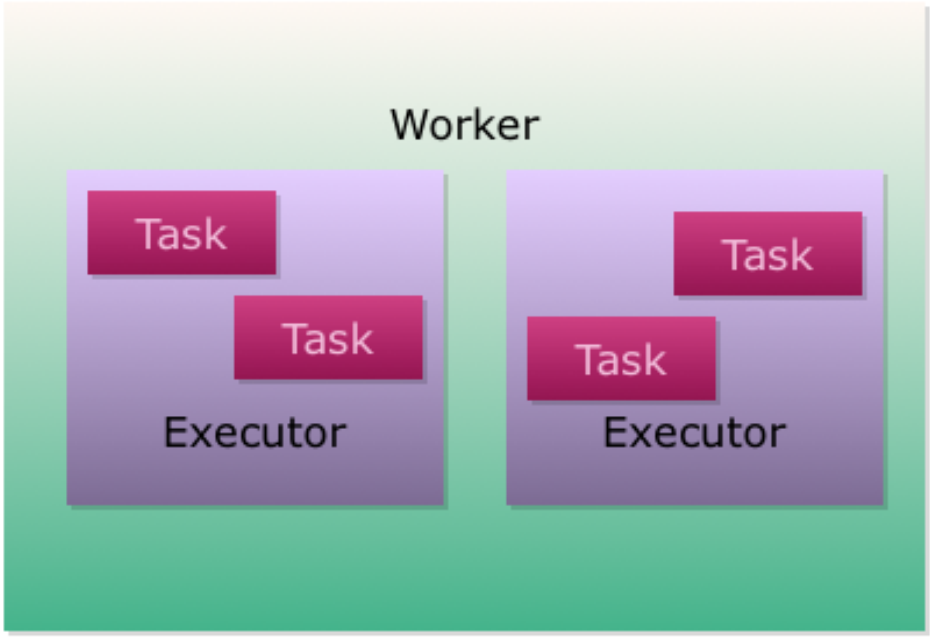
- The main method of an application
  - It's where the **`SparkSession`** is created
  - Establishes the connection to the cluster
  - Creates a DAG (Direct Acyclic Graph) of operations
- Connects to cluster manager to allocate resources
  - Acquires **executors** on worker nodes
  - Sends app code to executors
  - Sends **tasks** for executors to run
- Driver should be close to the worker nodes
  - Its location is independent of Master/Slave
  - Must be in a network addressable by Workers.
  - Preferably on the same LAN

Licensed for personal use only for Fernando K <fernando_kruse@dell.com> from Machine Learning at Dell Brazil (QE) @ 2019-03-12

Workers/Executors/Tasks

- Spark cluster has multiple workers
- Each worker can run multiple executors
- Each executor can run multiple tasks





Executors and Tasks

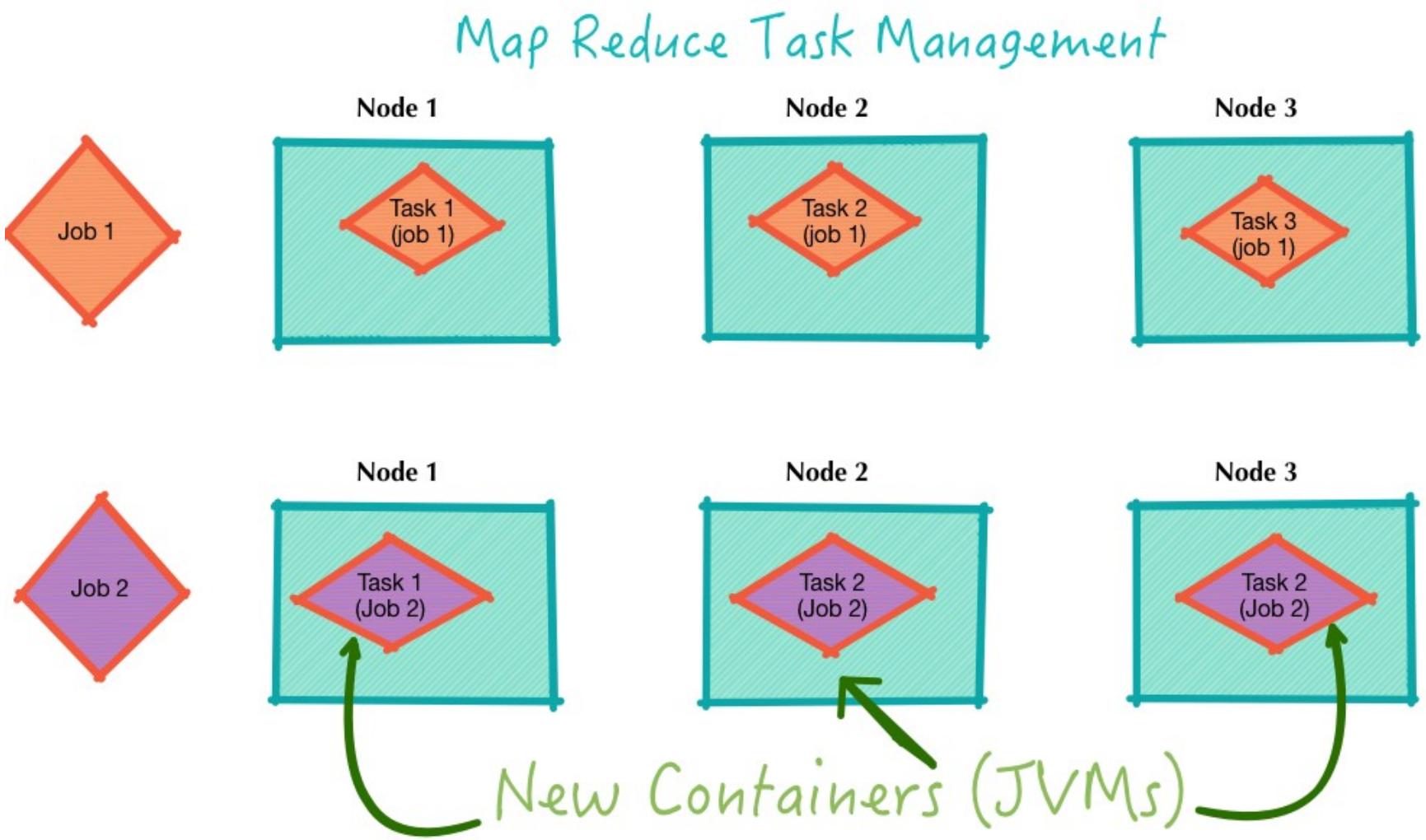
- **Executors**
  - Processes that run computations and store data
  - Each app gets its own executors
  - Launched at application startup, run for duration of the app
  - JVM containers (tasks from different apps in different JVM)
  - Execute tasks (in threads)
  - Provide memory for cache storage
- **Tasks**
  - "Smallest" execution units
  - Process data in partitions
  - Takes into account "data locality"
  - Runs as "threads" within executor JVM

Licensed for personal use only for Fernando K <fernando_kruse@dell.com> from Machine Learning at Dell Brazil (QE) @ 2019-03-12

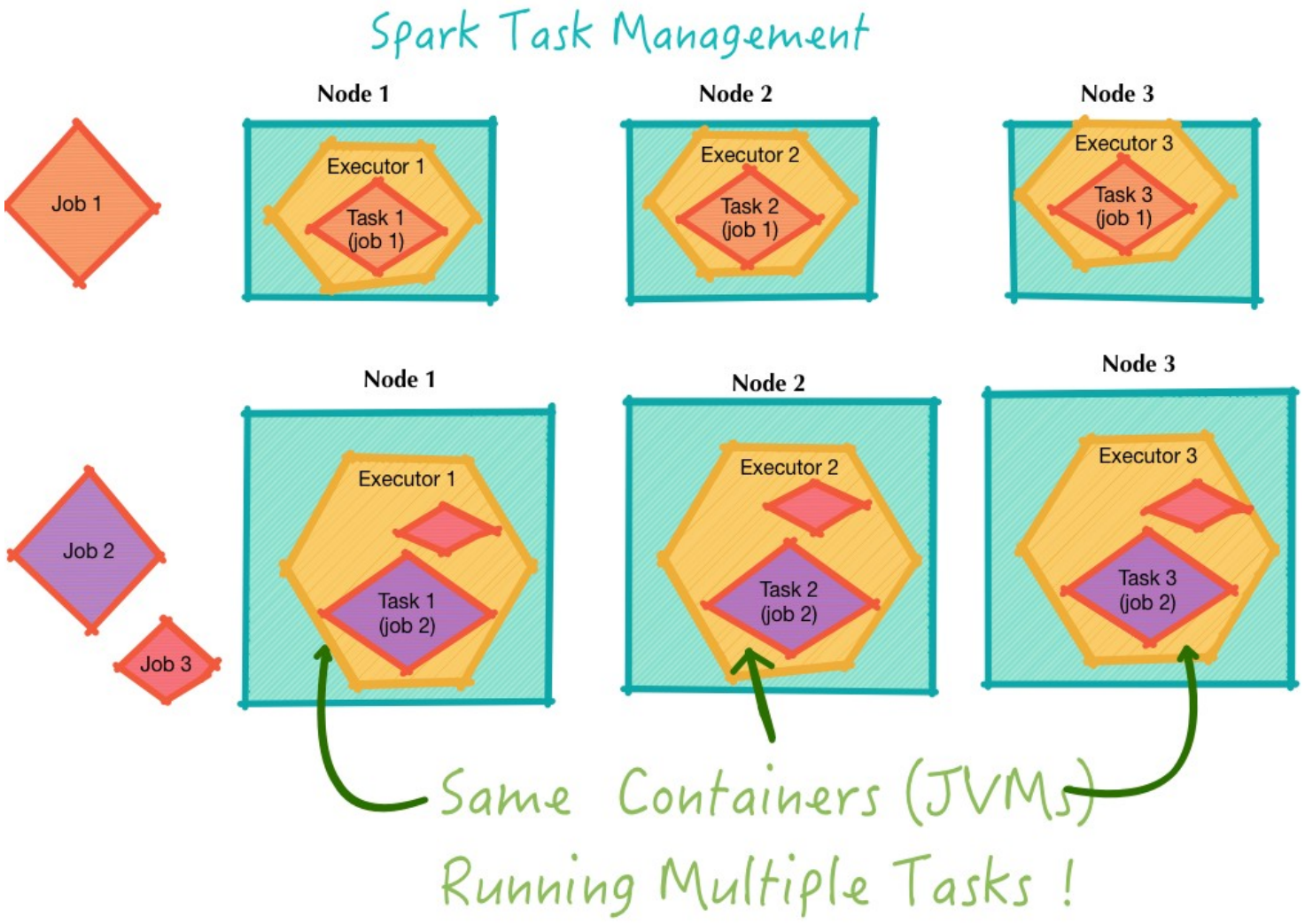
Spark Vs. Map Reduce

- Spark is much faster than Map Reduce
  - On disk data : 2x - 10x faster
  - In memory data : 100x faster !
- Spark has a different execution model than MR
- In MapReduce
  - Each job is run within its own ‘container’ (JVM)
  - Once the job is done, container is terminated
  - For another job, new containers get started, and run the new tasks.
  - This is expensive (starting a new JVM for each job)
- Spark
  - Containers (called ‘executors’) are ‘long lived’ (aka not terminated between jobs) (even when not running tasks)
  - New task startup is very fast (no warm up!)
  - Task switching is very fast too.
- Map Reduce is considering this approach too
  - LLAP : Long Live and Process
    (“Long Live and Prosper” ☺)

Map Reduce Task Management

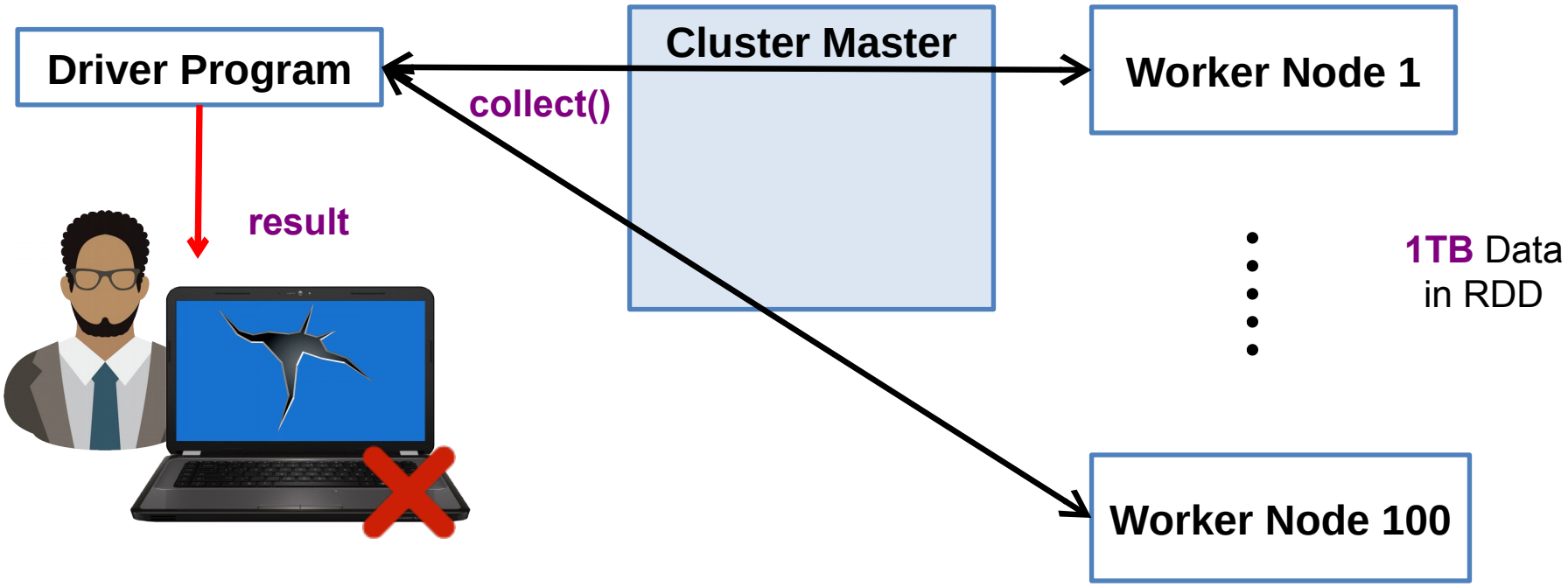


Spark Task Management



Driver Memory vs. Executor Memory

- Driver memory is generally small
- Executor memory is where data is cached–can be big
- `RDD.collect()` will send data to driver
  - Large collections will cause out-of-memory error in driver
  - Find a different way!



Spark & YARN

Core API

Building and Running Applications

➔ **Spark and YARN**

Application Lifecycle

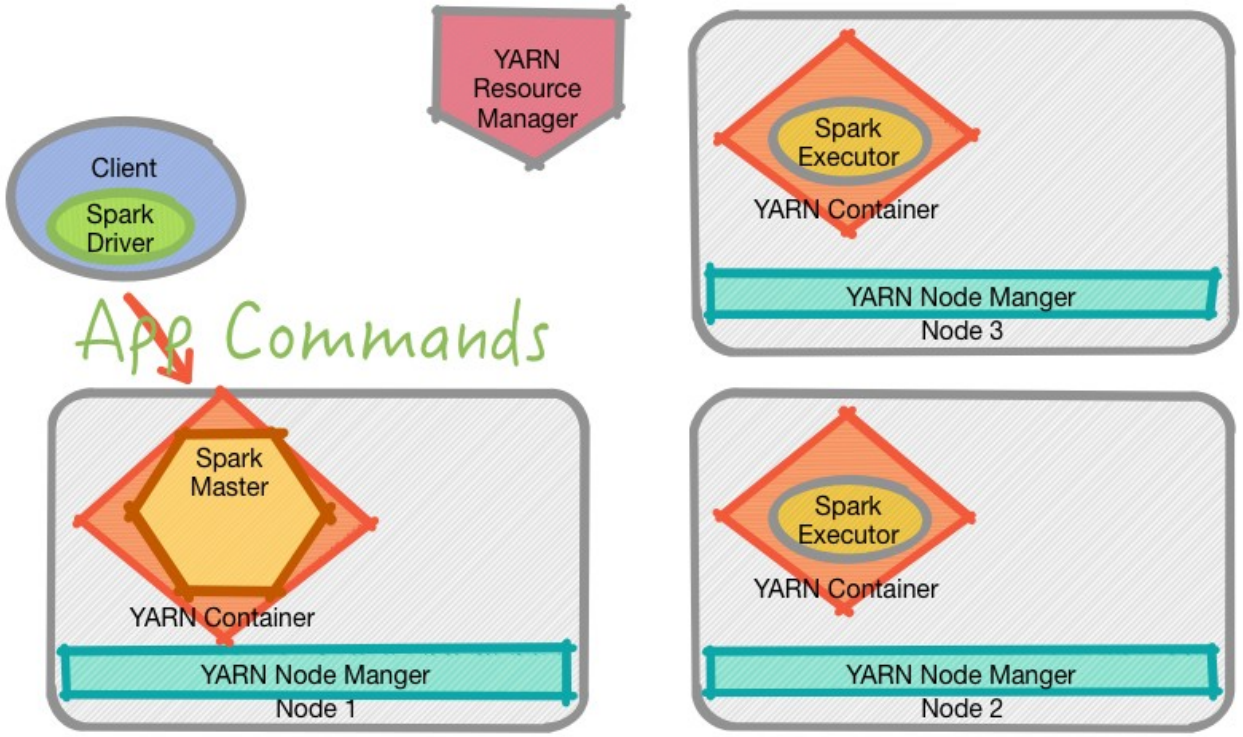
Logging & Debugging

Licensed for personal use only for Fernando K <fernando_kruse@dell.com> from Machine Learning at Dell Brazil (QE) @ 2019-03-12

Spark & YARN

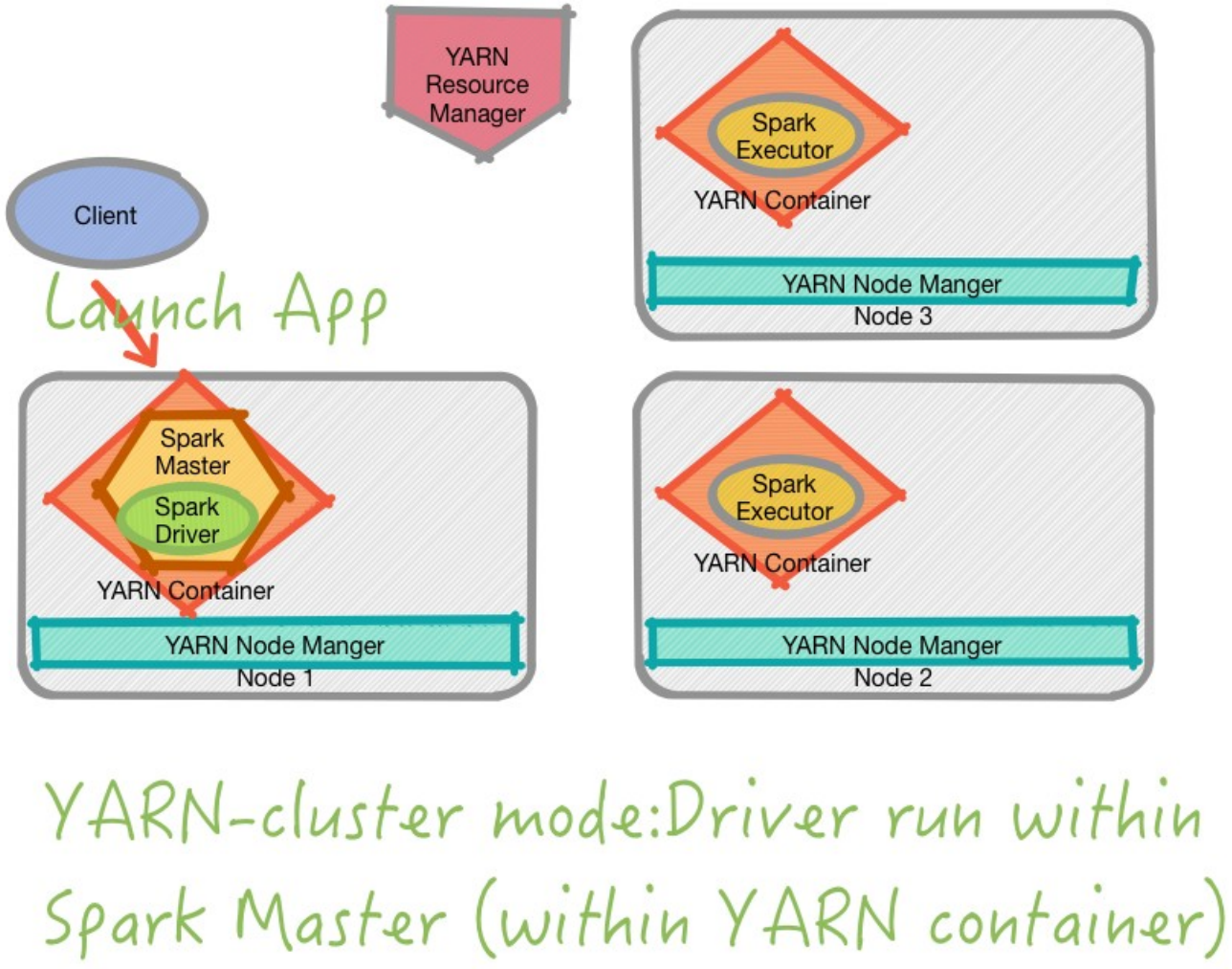
- Why run Spark on YARN
  - Deploy on existing Hadoop infrastructure
  - One Hadoop cluster can support multiple applications (MapReduce / Spark / HBase ..etc)
  - YARN provides process prioritization / isolation ..etc.
- 2 Modes : YARN-cluster, YARN-client
- YARN-client
  - Suited for interactive applications (spark-shell)
  - Development / debugging
- YARN-cluster
  - Production runs
  - Not meant for 'interactive' applications
  - Failure tolerant (if driver crashes, YARN will restart it – the container -- automatically)

Spark in YARN Client Mode



YARN-client mode: Driver run in client (outside YARN container)

Spark in YARN Cluster Mode



Running Spark in YARN

- Run mode is controlled by 'deploy-mode' flag
  - Client
  - cluster

Client Mode

```
spark-submit --class org.apache.spark.examples.SparkPi
--master yarn --deploy-mode client
SPARK_HOME/lib/spark-examples.jar 10
```

Cluster Mode

```
spark-submit --class org.apache.spark.examples.SparkPi
--master yarn --deploy-mode cluster
SPARK_HOME/lib/spark-examples.jar 10
```

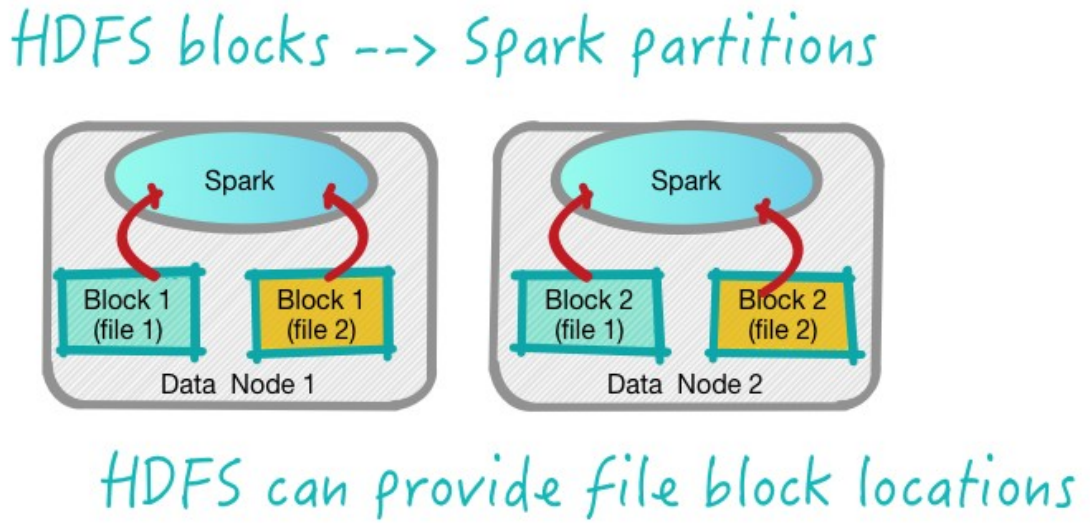
Spark Cluster Modes

	Standalone	YARN client	YARN cluster
Drivers runs in	Client	Client	App Master
Who requests resources	Client	App Master	App Master
Who starts executor process	Spark Slave	YARN Node Manager	YARN Node Manager
Persistent services	- Spark Master - Spark workers	- YARN Resource Manager - YARN Node Managers	- YARN Resource Manager - YARN Node Managers
Supports interactive applications like Spark-shell	Yes	Yes	No

Spark & HDFS

- Spark can natively read / write data to HDFS
- HDFS can also provide 'location hints' for data, so Spark can do 'data local' processing.
  - → faster processing (no network IO)
- HDFS is a high-throughput distributed file system

```
val logs = sc.textFile("hdfs://namenode:9000/data/*.log")
logs.count
```



Building and Running Applications

Core API

➔ **Building and Running Applications**

Application Lifecycle

Logging & Debugging

Licensed for personal use only for Fernando K <fernando_kruse@dell.com> from Machine Learning at Dell Brazil (QE) @ 2019-03-12

Core API - Scala

➔ **Core API**

Building and Running Applications

Application Lifecycle

Logging & Debugging

Basic Driver Code (Scala)

```
import org.apache.spark.sql.SparkSession

// Basic Spark App (Scala)
object TestApp{
  def main(args: Array[String]) {
      val spark = SparkSession.builder().

appName("Test").
                          getOrCreate()

      val f = spark.read.textFile("file")
      println ("# lines: " + f.count)

    spark.stop()
  }
}
```

1. Create **`SparkSession`** instance
2. Configure any parameters using **builder**
3. Code operations as needed—no different from previous

SparkSession.builder ( )Configuration Properties

```
val spark  = SparkSession.builder.getOrCreate()

val spark  =
SparkSession.builder.x().y().z().getOrCreate()
```

API	Description	Example
appName()	Sets the application name (shows up in UI)	SparkSession.builder.appName(“TestApp”)
master()	Mater URL	SparkSession.builder.master(“local”) (See next page for options)
config()	Set a property for the app	SparkSession.builder.config (“cassandra.host”, “host1”)

Builder Configuration Properties

- `master(master:String)`: Master URL to connect to

Master Key	Description	Example
Local		
local	localhost with a single CPU core	“local”
local[N]	Run on localhost with N CPU cores	“local[4]”
local[*]	Run on localhost with all CPU cores	“local[*]”
Distributed		
spark://host:port	Spark master (running stand alone)	spark:// host1:7077
mesos:// host:port	Spark master (running on Mesos)	mesos:// host1:5050
Yarn	Running on YARN	“yarn”

Configuration Property Names

- Can also set configuration via named properties, including:
  - **`spark.master`**: Master URL (same as `setMaster()`)
  - **`spark.app.name`**: App name (same as `setAppName()`)
- Can pass properties to `spark-submit` using **`--conf`**

```
./bin/spark-submit … \
          --conf
spark.master=spark:/1.2.3.4:7077
```

- spark-submit will also read configuration options in the file *<spark>/conf/**spark-defaults.conf***
  - In standard `key=value` properties file format
  - Precedence order (highest to lowest): (1) Properties set directly on `SparkSession`, (2) props passed to spark-submit, (3) *spark-defaults.conf*

SparkSession

- Use builder to create one

```
val spark =
SparkSession.builder.getOrCreate()
```

- Can create dataset

```
spark.createDataset()
```

- Use read method

```
spark.read.parquet()
```

- Access underlying SparkContext

```
spark.sparkContext
```

- Execute SQL

```
spark.sql("select * from t1")
```

- Only one active SparkSession per JVM

  `stop()` the active one before creating a new one

Review Documentation

Mini-Lab

Browse to **http://spark.apache.org/docs/latest/**

- On the top menu bar, go to **API Docs | Scala**
- In the left hand pane, type in `SparkSession` in the filter box
  - Review its docs for a short time
- Now filter for `RDD`
  - Review the RDD docs
  - Notice the operations relating to the internal API

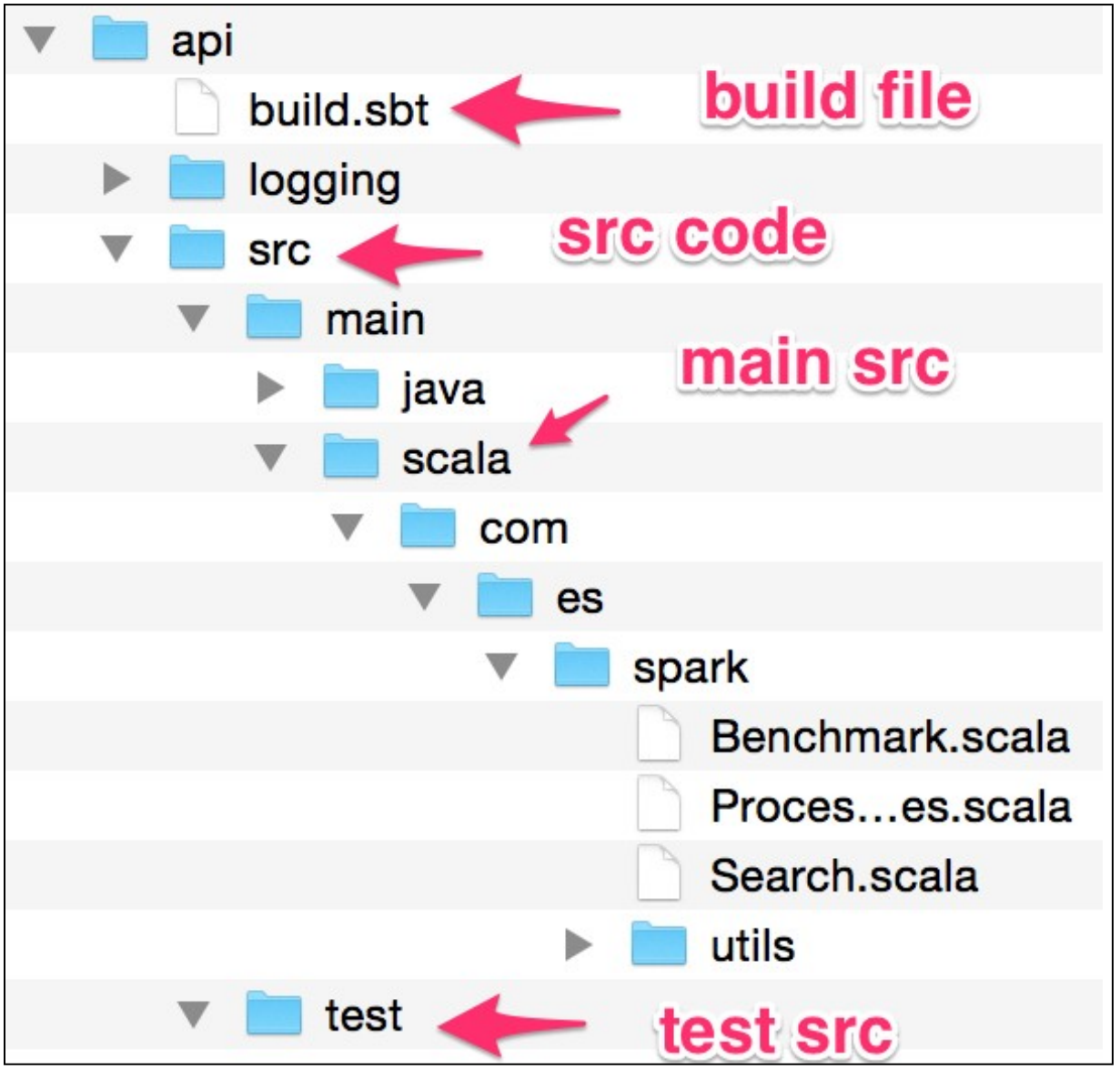
Tools for Building/Coding

- **Python** – nothing extra
- **Scala - sbt**: Simple Building Tool
  - http://www.scala-sbt.org/
- **Java - maven**: Just need the Spark dependencies–for example:

```
<dependency>
  <groupId>org.apache.spark</groupId>
  <artifactId>spark-core_2.10</artifactId>
  <version>1.2.0</version>
</dependency>
```

- **Editors and IDEs**
  - **Scala IDE**: Eclipse-based Scala IDE
  - **IntelliJ**: Excellent Scala support, fast/incremental compile
  - **Sublime**: Sophisticated text editor–full Scala support

Application Layout (Scala)



- Uses maven layout by default

Scala - build.sbt

- The build file for sbt
  - This one sets the app name, app version, and Scala version
  - It then configures dependencies

```
name:= "TestApp"

version:= "1.0"

scalaVersion:= "2.11.7"

// ++= means concatenate sequence of dependencies
// %% means append Scala version to next part
libraryDependencies ++= Seq(
  "org.apache.spark" % "spark-core" % "2.2.0” % “provided”
)

// need this to access files on S3 or HDFS
// += means just append the dependency
libraryDependencies += "org.apache.hadoop" % "hadoop-client" %
"2.7.0" exclude("com.google.guava", "guava")
```

Scala - Compiling Code

- *build.sbt* generally in project root dir
  - Same purpose as *pom.xml* for Maven
- Automatically downloads dependencies
- sbt commands
  - `sbt compile`
  - `sbt package` –builds a jar
  - `sbt assembly` –builds a “fat” jar with all the dependencies
  - `sbt clean` –cleans up all generated artifacts
- To re-build completely

```
sbt clean package
```

  - The first run takes a few minutes to download all dependencies
  - Go get some coffee ☺

spark-submit (Scala)

```
./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  x.ProcessFiles \
   target/scala-2.11/testapp.jar 1G.data
```

- `<spark>/bin/spark-submit` can launch apps on the cluster
  - Can be used with all supported cluster managers
  - See next page for options explanations
- At bottom, we submit to a standalone manager, set executor memory, and pass an argument (a file name)

spark-submit (Scala)

Flag	Description	Example
--master <master url>	Master url	--master Spark://host1:7077
--name <app name>	Application Name	--name TestApp
--class <main class>	Main class	--class my.TestApp
--driver-memory <val>	Memory for app driver (default 512M)	--driver-memory 1g
--executor-memory <val>	Memory for executors (more important!)	--executor-memory 4g
--deploy-mode <deploy-mode>	Deploy driver to worker (cluster) or run locally (client)	--deploy-mode cluster
--conf <key>=<value>	Spark Config Property	
--help	Print out all options	

Scala - Lean/Fat Jars, Conflicts

- 'sbt package' produces **lean** jar
  - Just our application code only
  - Good for apps with no dependencies

- '**`sbt assembly`**' will produce a **fat** jar
  - Pulls in all dependencies
  - Good if we have dependencies using custom versions

- **Conflict** Problem
  - System comes with V1.0 of package X, my app needs V 2.0
  - One solution: **Produce a fat jar** with the required dependency
  - See next slide for sbt example

Scala - sbt Assembly Example

```
import AssemblyKeys._

name:= "TestApp"

version:= "1.0"

scalaVersion:= "2.11.7"


libraryDependencies ++= Seq(
  "org.apache.spark" %% "spark-core" % "2.2.0” % “provided”
)

// need this to access files on S3 or HDFS
libraryDependencies += "org.apache.hadoop" % "hadoop-client" % "2.7.0"
exclude("com.google.guava", "guava")

assemblySettings

mergeStrategy in assembly:= {
  case m if m.toLowerCase.endsWith("manifest.mf")          => MergeStrategy.discard
  case m if m.toLowerCase.matches("meta-inf.*\\.sf$")      => MergeStrategy.discard
  case "log4j.properties"                                  => MergeStrategy.discard
  case m if m.toLowerCase.startsWith("meta-inf/services/") =>
MergeStrategy.filterDistinctLines
  case "reference.conf"                                    => MergeStrategy.concat
  case _                                                   => MergeStrategy.first
}
```

Core API - Python

➔ **Core API**

Building and Running Applications

Application Lifecycle

Logging & Debugging

Licensed for personal use only for Fernando K <fernando_kruse@dell.com> from Machine Learning at Dell Brazil (QE) @ 2019-03-12

Basic Driver Code (Python)

```
from pyspark.sql import SparkSession

spark = SparkSession.builder.
        appName("mysparkapp").getOrCreate()

f = spark.read.text("README.md")
count = f.count()

print("###  number of lines", count)

spark.stop()  # close the session
```

1. Create **`SparkSession`** instance
2. Configure any parameters using **builder**
3. Code operations as needed—no different from previous

spark-submit (Python)

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

```
$  spark-submit --master spark://localhost:7077 \
   --executor-memory 4G  process-files.py 1G.data
```

- `<spark>/bin/spark-submit` can launch apps on the cluster
  - Can be used with all supported cluster managers
  - See next page for options explanations
- At bottom, we submit to a standalone manager, set executor memory, and pass an argument (a file name)

spark-submit (Python)

Flag	Description	Example
--master <master url>	Master url	--master Spark://host1:7077
--py-files X	Additional python files needed. Could by .py or .zip (with multiple files)	--py-files another-file.py
--driver-memory <val>	Memory for app driver (default 512M)	--driver-memory 1g
--executor-memory <val>	Memory for executors (more important!)	--executor-memory 4g
--deploy-mode <deploy-mode>	Deploy driver to worker (cluster) or run locally (client)	--deploy-mode cluster
--conf <key>=<value>	Spark Config Property	
--help	Print out all options	

Python Dependencies

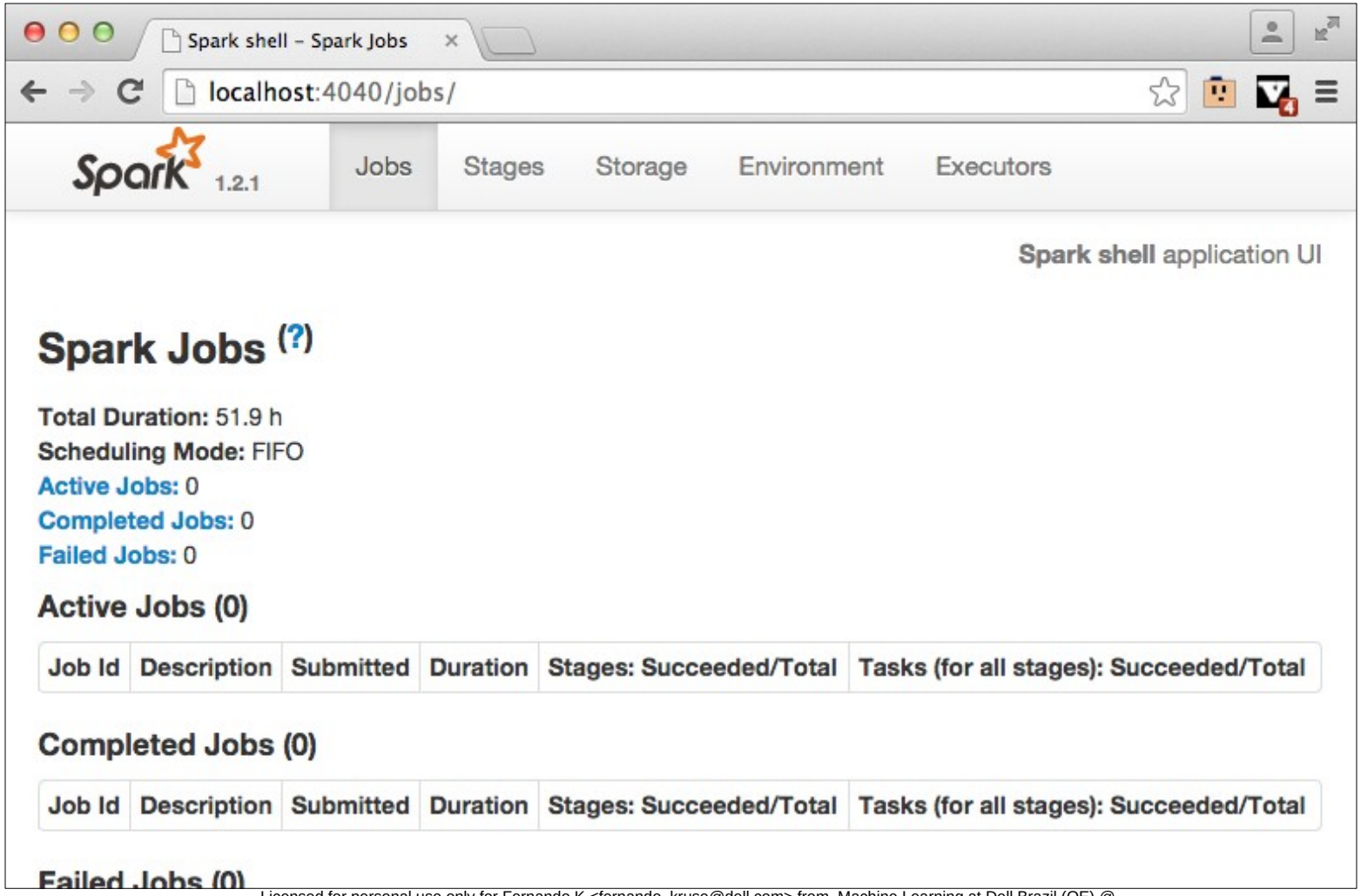
- If the application depends on other python libraries (like numpy …etc) they needed to be installed on worker machines **before hand**

- Spark does NOT install the packages automatically

Licensed for personal use only for Fernando K <fernando_kruse@dell.com> from Machine Learning at Dell Brazil (QE) @ 2019-03-12

Application UI

- The driver provides a Web UI with application details
  - At <driver-node>:4040 (e.g., localhost:4040 for local)



Lab Tip: Editing Files ON VM

- **Option 1 (Easiest) – Using VNC desktop Instructor please demo the following steps**
  - Logging into VNC
  - Opening Sublime editor (or any other editor)
  - Opening 'spark-labs' directory
  - And editing the file

- **Option 2 : Use vi or nano editors**

```
$     cd  ~/spark-labs/
$     vi  file_name_to_edit
# or
$     nano  file_name_to_edit
```

Lab: Spark Job Submission

- **Overview**:
  In this lab, we will build and run a simple Spark app
  - We'll build with sbt
  - We'll submit the job to the Spark cluster using spark-submit
- **Builds on previous labs**:
  Lab for general setup
- **Approximate time**:
  20-30 minutes
- **Instructions**:
  Follow: **5-api/5.1-submit.md**
- **Solution**:
  /spark/spark-solutions/5-api

Logging and Debugging

Core API

Building and Running Applications

Application Lifecycle

➔ **Logging & Debugging**

Licensed for personal use only for Fernando K <fernando_kruse@dell.com> from Machine Learning at Dell Brazil (QE) @ 2019-03-12

We Will Review Some Common Techniques

- There are some very easy ways to get information about what Spark is doing
  - And how your job is going (or how it has gone)
- We will cover:
  - `RDD.toDebugString()` to visualize an RDD
  - The Web-based UI for the driver and master
  - Configuring logging

toDebugString()

- **RDD.toDebugString()**: A description of this RDD and its recursive dependencies for debugging
  - Very useful to visualize what is happening in a Spark application
  - We illustrate this below using a simple word count program

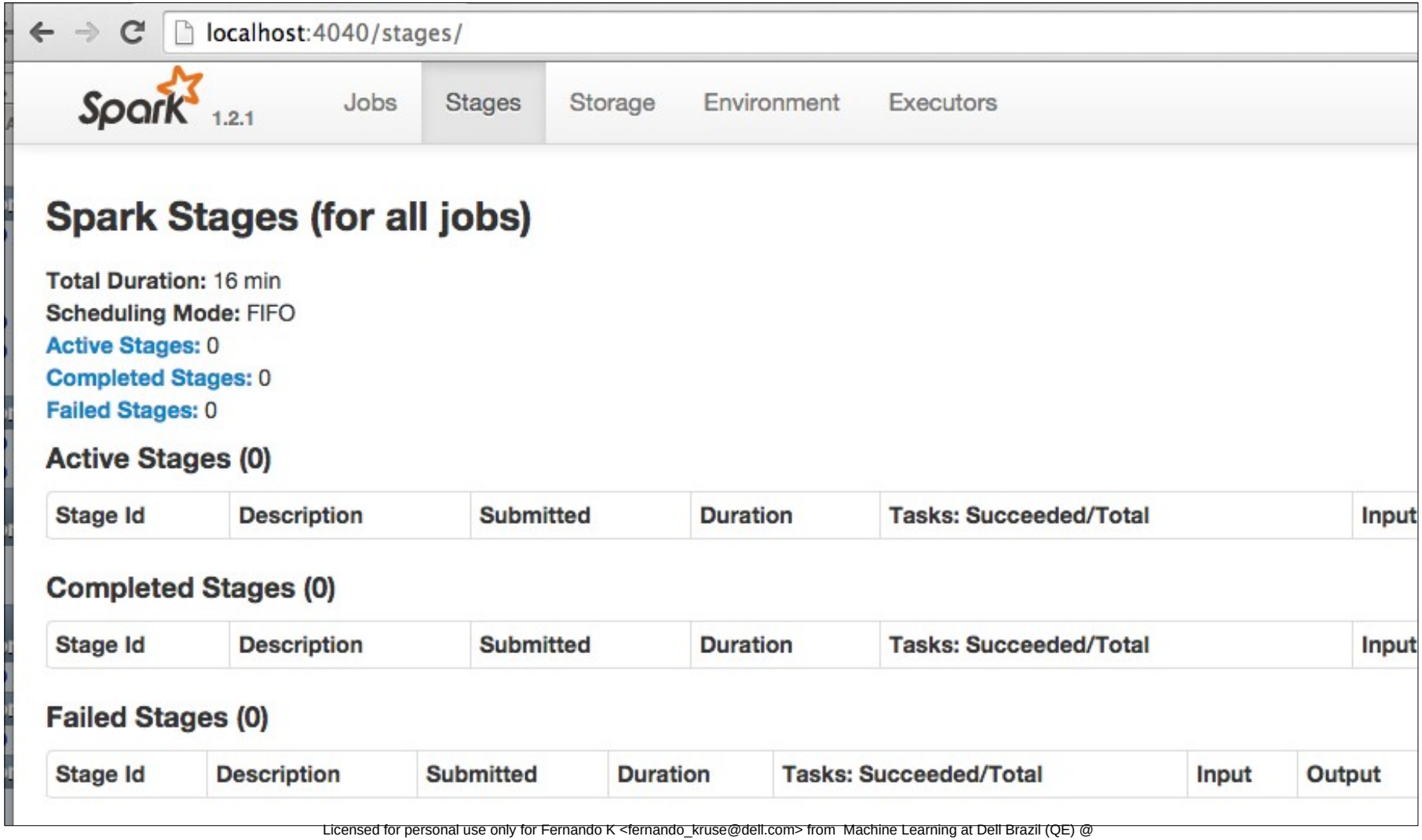
```
scala> val line = "apple pear apple grape pear apple"
scala> val wordPairsRDD = sc.parallelize(line.split("\\
s+")).map(word => (word,1))
scala> val countsRDD = wordPairsRDD.reduceByKey(_ + _)

scala> wordPairsRDD.toDebugString
res2: String =
(8) MapPartitionsRDD[4] at map at <console>:23 []
 |  ParallelCollectionRDD[3] at parallelize at <console>:23 []

scala> countsRDD.toDebugString
res0: String =
(8) ShuffledRDD[2] at reduceByKey at <console>:16 []
 +-(8) MappedRDD[1] at map at <console>:14 []
    |  ParallelCollectionRDD[0] at parallelize at <console>:14 []
```

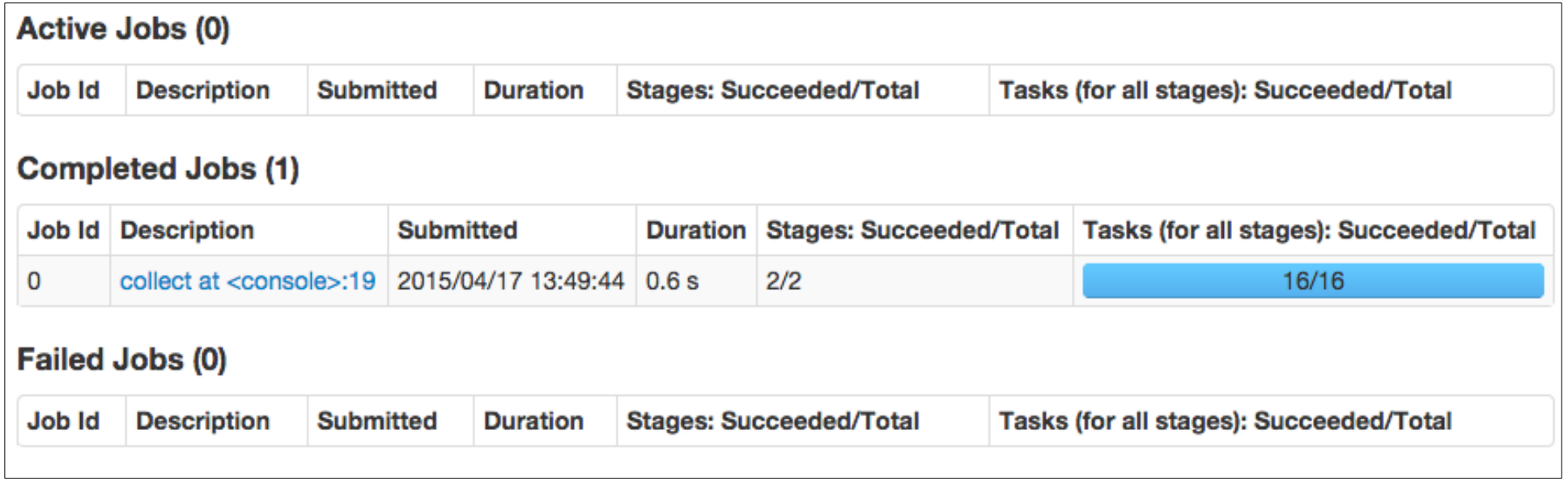
Web UI

- We've seen the Web UI from the driver earlier (at port 4040)
  - It can be used to monitor ongoing and completed jobs



Information from Web UI

- Below is the UI after executing `countsRDD.collect`
  - Note how there is one completed job
- Clicking on the Job Description brings you to a detail page
  - See next slide



Active Jobs (0)

Job Id	Description	Submitted	Duration	Stages: Succeeded/Total	Tasks (for all stages): Succeeded/Total

Completed Jobs (1)

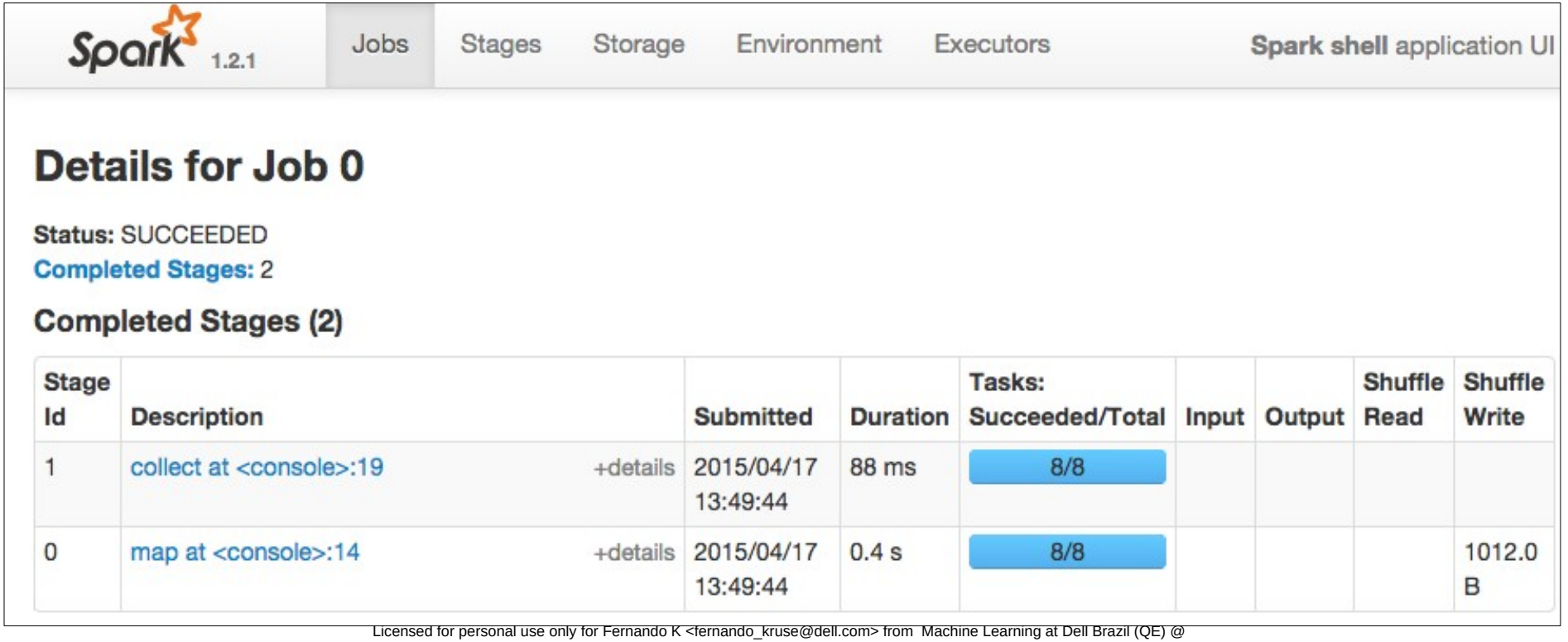
Job Id	Description	Submitted	Duration	Stages: Succeeded/Total	Tasks (for all stages): Succeeded/Total
0	collect at <console>:19	2015/04/17 13:49:44	0.6 s	2/2	16/16

Failed Jobs (0)

Job Id	Description	Submitted	Duration	Stages: Succeeded/Total	Tasks (for all stages): Succeeded/Total

Viewing Stages

- Below, you can see the stages of the job
  - There are two—one for the map, and one for the reduce (which requires a shuffle)
  - Clicking on the stage description will bring up a detail page for it



Stage Detail

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

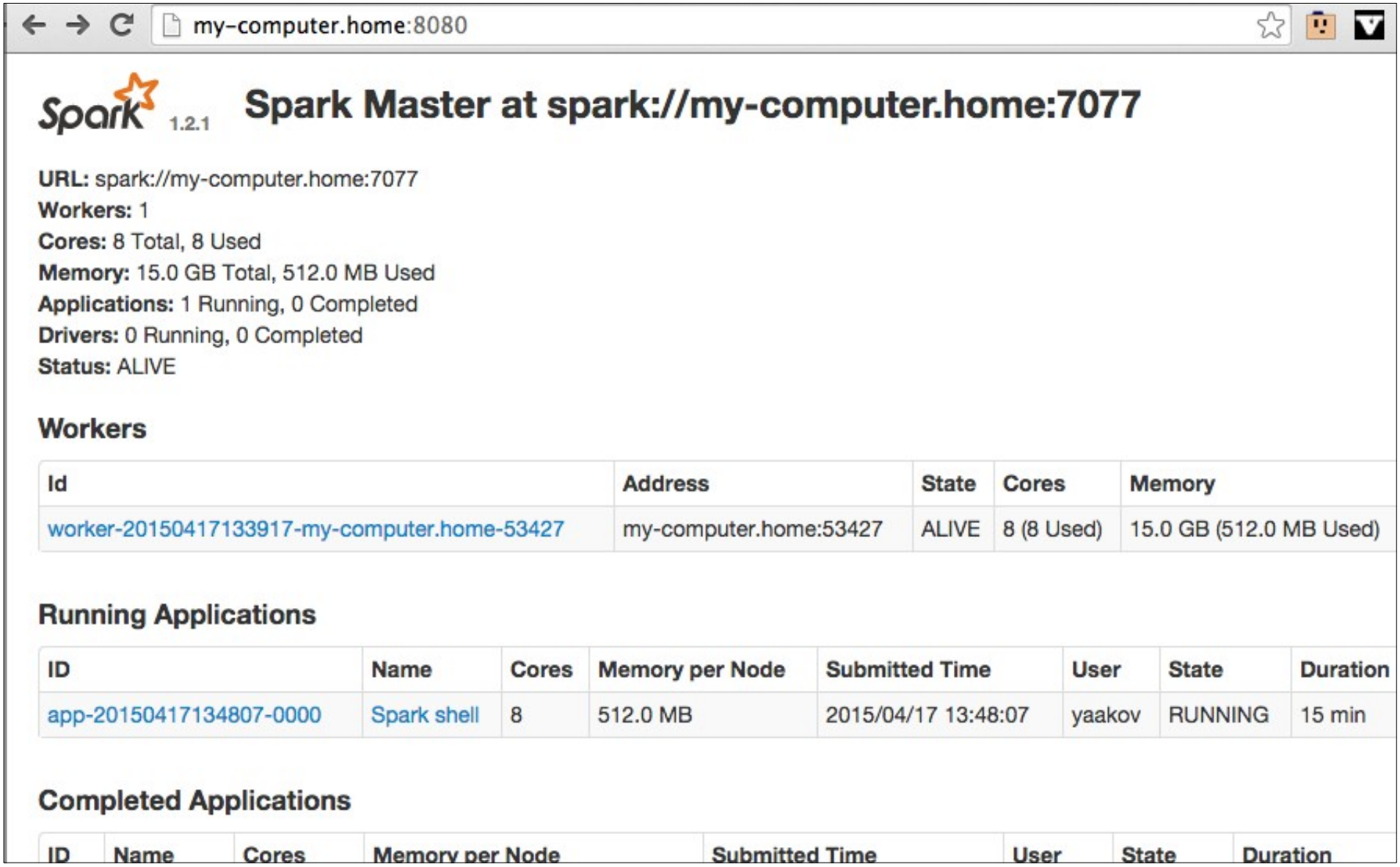
Aggregated Metrics by Executor

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

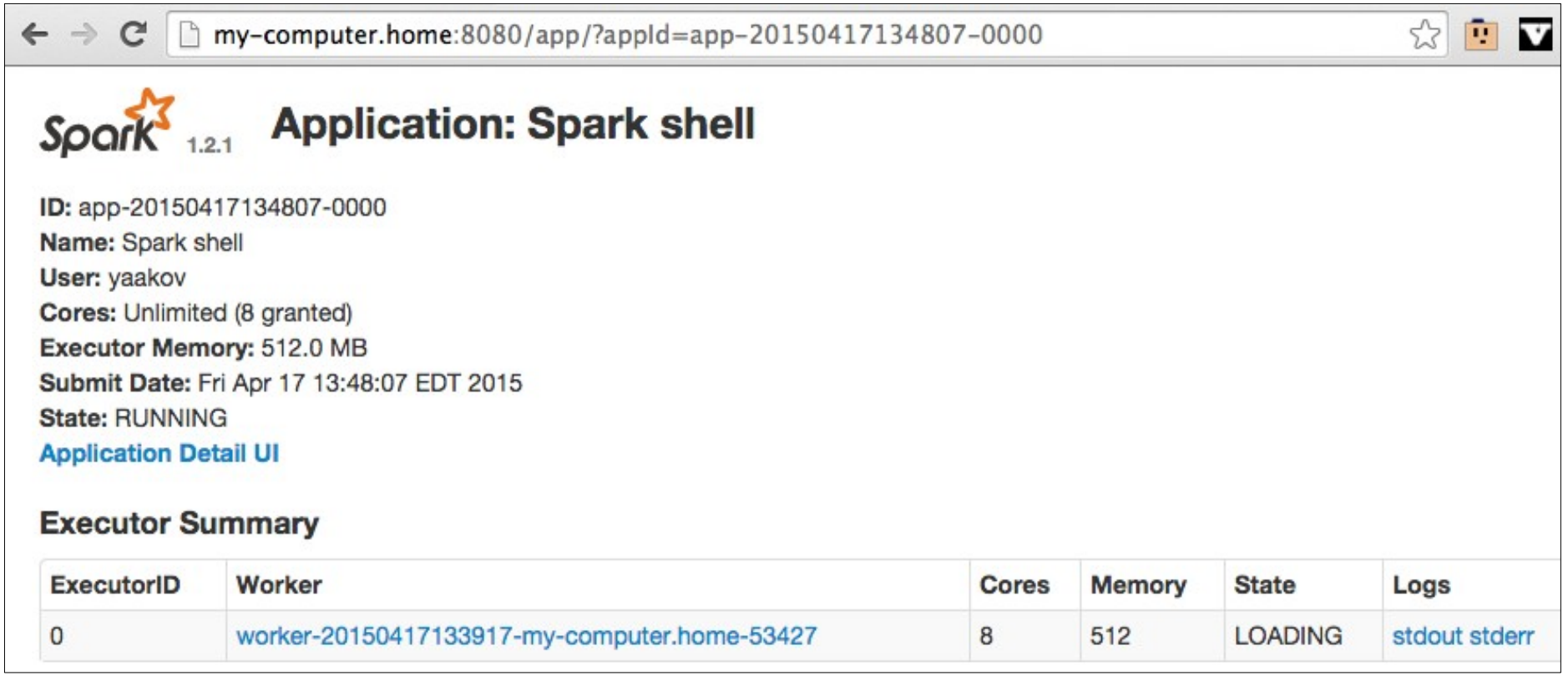
Index	ID	Attempt	Status	Locality Level	Executor ID / Host	Launch Time	Duration	GC Time	Errors
0	8	0	SUCCESS	PROCESS_LOCAL	0 / my-computer.home	2015/04/17 13:49:44	59 ms		
1	9	0	SUCCESS	PROCESS_LOCAL	0 / my-computer.home	2015/04/17 13:49:44	58 ms		
3	11	0	SUCCESS	PROCESS_LOCAL	0 / my-computer.home	2015/04/17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

Master UI: <master-host>:8080



Master UI: Application Detail

- This is the detail page for the Spark shell
  - Note how you can access stdout and stderr directly



Logging

- Master logs appear in *<spark>/logs*
  - Log files are named after the user and computer name
  - E.g., **spark-student-org.apache.spark.deploy.master.Master-1-my-computer.home.out**
- Application logs appear in *<spark>/work*
  - In a subdirectory created when the app starts
  - For each app, there is a file for stdout and stderr logging
  - The logging output is also visible in the Master UI, as seen earlier
    - But not all output (E.g., INFO output is not visible there)

Customizing Logging

- Logging can be customized by creating the file *conf/log4j.properties*
- There is a template, *conf/log4j.properties.template,* you can copy
- Below, we illustrate how to change the root logger level to WARN
- This reduces some of the copious logging that Spark produces
- This file will be detected and used automatically
- You can also use the `--files` option to spark-submit to send this file along with your app jar
- Make sure to distribute the log4j.properties file to all nodes

```
# 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 …
```

Turning Off Driver Logging

- Even modifying conf/log4j.properties might not have any effect on driver logging!
  - This is the console output you see when submitting a job
- Need to explicitly specify the log4j.location

```
$ ./bin/spark-submit –master spark://localhost:7077 \
   --driver-class-path  logging/
   --class  com.es.spark.ProcessFiles \
   target/scala-2.10/testapp.jar 1G.data
```

[Bonus] Lab: MapReduce Application

- **Overview**:
  In this lab, we will run a MapReduce application by submitting it with spark-submit

- **Builds on previous labs**:
  Lab for general setup

- **Approximate time**:
  20-30 minutes

- **Instructions**:
  Follow: **5-api/5.2-maprduce.md**

Review Questions

1. Does Spark run code as interpreter or does it require compiled applications?
2. What is Application driver?
3. How is Application driver different from Spark master?
4. What are executors and tasks?