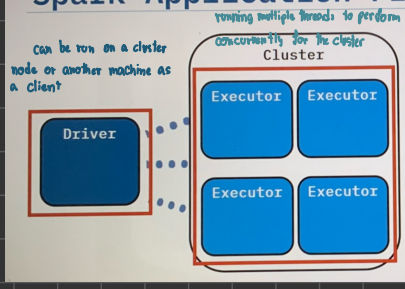
Apache Spark Architecture

Spark Application Processes



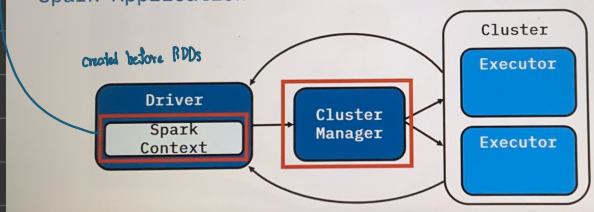
A Spark Application has two main processes:

- Driver Program: a single process that creates work for the cluster and send it to a cluster
- Executors: multiple processes throughout the cluster that do the work in parallel (work independently)

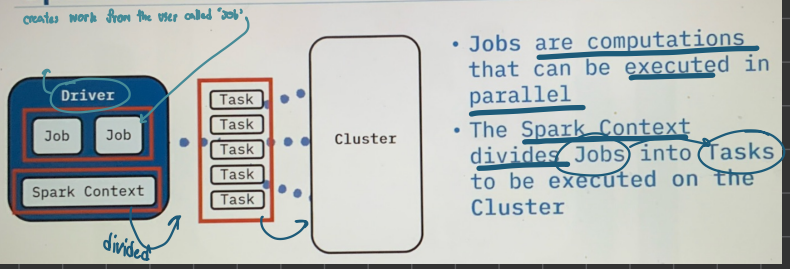
What is the Spark Context

Spark Context: starts when the application launches and must be created in the driver

- Communicates with the Cluster Manager
- Is defined in the Driver, with one Spark Context per Spark Application (any data frame or RDD created under the context)

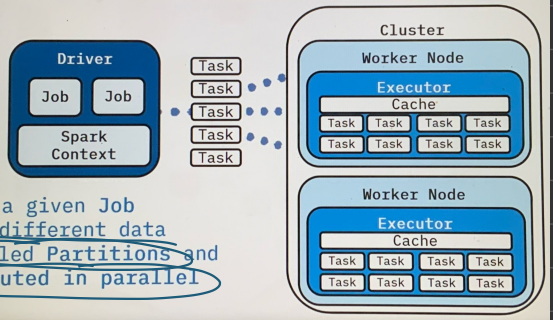


Spark Jobs



- Jobs are computations that can be executed in parallel
- The Spark Context divides Jobs into Tasks to be executed on the Cluster

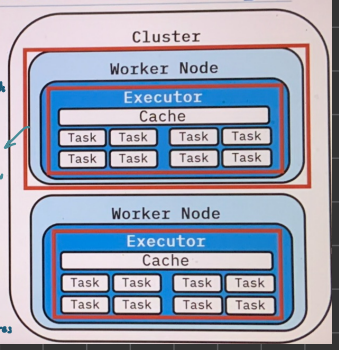
Spark Tasks



Tasks from a given Job operate on different data subsets called Partitions and can be executed in parallel

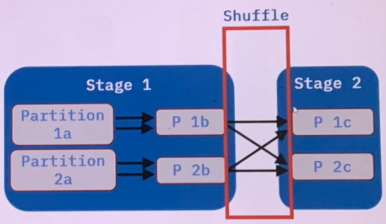
Spark Executors run tasks from a job

- A Worker is a cluster node that can launch executor processes to run tasks
- Each Executor is allotted a set number of cores that each run one task at a time (utilizes a set portion of local resources as memory and compute cores, one core/task per available core)
- Increasing Executors and cores increases cluster parallelism (When a task finishes, the executor gets the results in a new RDD)
- Ideally, limit Executor x core combinations to total cores per node – the example here shows 1 x 8 (Tasks run separate threads until all cores are used)



Spark Stages and Shuffles

- A Stage is a set of tasks within a job that can be completed on the current local data partition
- A Shuffle marks the boundary between Stages
- Stages connect to form a dependency graph

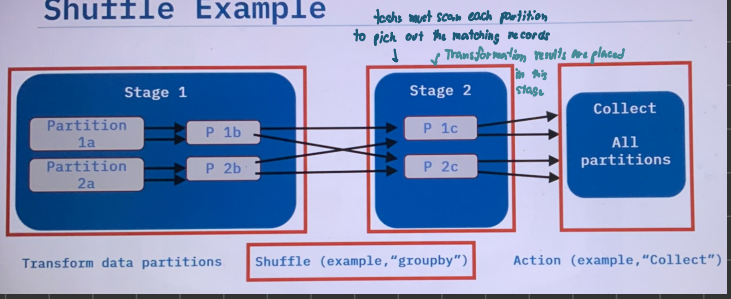


Why Shuffle Data

A Shuffle is:

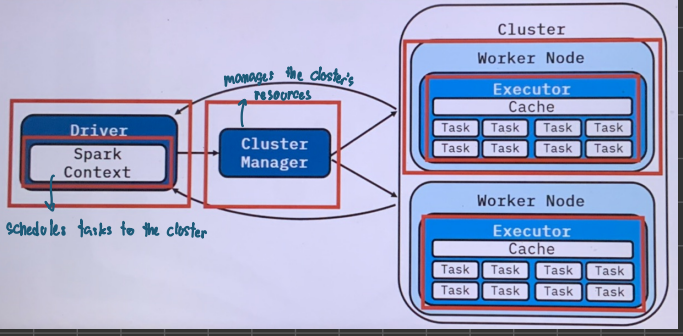
- Costly – requiring data serialization, disk and network I/O
- Necessary when an operation requires data outside the current partition of the task
- How Spark re-distributes the dataset across the cluster

Shuffle Example



Recap

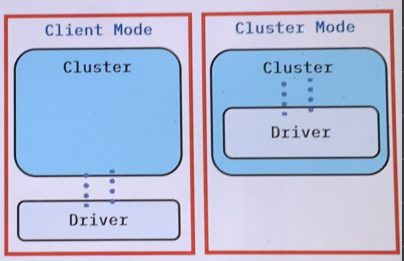
Apache Spark Architecture



Driver Deploy Modes

There are two deploy modes:

- Client Mode – the application submitter launches the driver process outside the cluster
- Cluster Mode – the framework launches the driver process inside the cluster



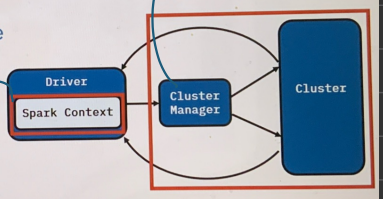
Overview of Apache Spark Cluster Modes

Spark Cluster Manager

Spark Cluster Manager:

- Communicates with a particular cluster to acquire resources for the running application
- Runs as a service outside the application and abstracts the cluster type

reserves executor cores and memory resources then transfer to execute processes to run

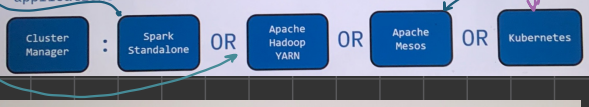


while an App. is running, the Spark context creates tasks and communicates to the cluster manager what resources are needed

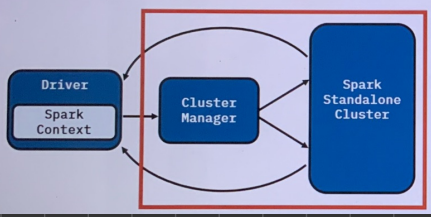
Types of Cluster Managers

Spark supports the following cluster managers:

- Spark Standalone – comes with Spark, best for setting up a simple cluster
- Apache Hadoop YARN (Yet Another Resource Negotiator) – cluster manager from the Hadoop project
- Apache Mesos – general-purpose cluster manager with additional benefits
- Kubernetes – open-source system for running containerized applications



Why use Spark Standalone



Spark Standalone benefits are:

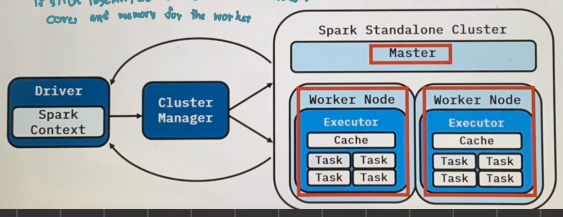
- Built into the Spark installation, so no additional dependencies to deploy
- Fastest way to setup a Spark cluster and get running

Spark Standalone components

Spark Standalone has two main components:

- Workers – run an Executor process to receive tasks
- Master – connects and adds workers to the cluster and keep track of them

if stick together, do not reserve all the node's cores and memory for the worker



How to set up Spark Standalone

1. Start the Master to output the Master URL

```
$ ./sbin/start-master.sh
```

2. Start Worker(s) with the Master URL

```
$ ./sbin/start-slave.sh spark://<master-spark-URL>:7077
```

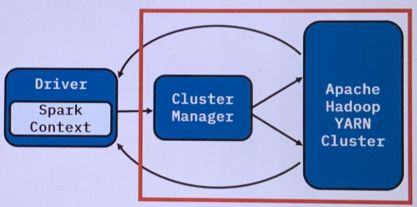
3. Launch Spark application on cluster by specifying the Master URL

```
$ ./bin/spark-submit \
    --master spark://<spark-master-URL>:7077 \
    <additional configuration>
```

Why use Apache Hadoop YARN

When choosing Apache Hadoop YARN, consider that it:

- Is general-purpose
- Supports many other big data ecosystem frameworks
- Requires its own configuration and setup
- Has dependencies, making it more complex to deploy than Spark Standalone



How to run Spark on existing YARN cluster

1. Specify the Master option `--master YARN` with 'spark-submit'

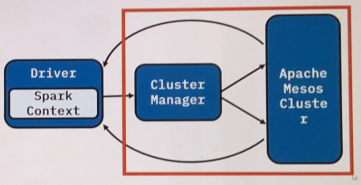
```
$ ./bin/spark-submit \
    --master YARN \
    <additional configuration>
```

2. Spark will automatically connect with YARN using Hadoop configuration

Why use Apache Mesos

Apache Mesos Cluster Managers can run Spark with other benefits, such as making partitioning:

- Scalable between many Spark instances
- Dynamic between Spark and other big data frameworks



To configure Spark to run on an Apache Mesos cluster manager see:

https://spark.apache.org/docs/latest/running-on-mesos.html

Why use Spark on Kubernetes

Kubernetes cluster managers can run containerized applications, making it easier to:

- Automate deployment
- Simplify dependency management
- Scale the cluster

How to run Spark on Kubernetes

To launch a Spark Application on Kubernetes use:

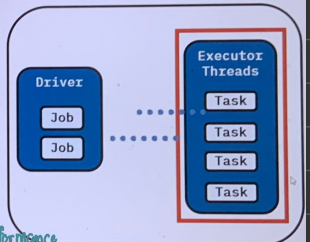
```
$ ./bin/spark-submit \
    --master k8s://https://<k8s-apiserver-host>:<k8s-apiserver-port> \
    <additional configuration>
```

Why use Local Mode

Spark can also be run in local mode which:

- Does not connect to a cluster, making it easy to get started
- Runs in the same process that calls 'spark-submit' and uses threads for running executor tasks
- Can be useful for testing or debugging a Spark application
- Runs a Spark application locally within a single process which can limit performance

local mode isn't designed for optimal performance



How to set up Local Mode

To run Spark in local mode, use the master option `--master local[#]` where `#` specifies the number of cores to use

```
# Launch Spark in local mode with 8 cores
$ ./bin/spark-submit \
    --master local[8] \
    <additional configuration>
```

Use an asterisk '*' to specify using all available cores

```
# Launch Spark in local mode with all available cores
$ ./bin/spark-submit \
    --master local[*] \
    <additional configuration>
```

How to Run an Apache Spark Application

What is 'spark-submit'

'spark-submit' script:

- Spark's Unified Interface for submitting applications
- Found in the 'bin/' directory
- Easily switches from local to cluster mode by changing a single argument
- Works the same way regardless of cluster manager type or application language

```
$ ./bin/spark-submit <config and options> <application files>
```

Using the 'spark-submit' Script

The `spark-submit` script will:

1. Parse command line arguments/options
2. Read additional configuration specified in `conf/spark-defaults.conf`
3. Connect to the cluster manager specified with the `--master` argument or run in local mode
4. Transfer application (JARs or Python files) and any additional files specified to be distributed and run in the cluster

Common 'spark-submit' Options

Option/Setting	Form	Mandatory ?
Tell Spark what cluster manager to connect with	'--master'	Yes
Specify the program entry point if using Java or Scala application	`--class <full-class-name>`	Yes
Set how the driver is deployed (Client or Cluster). Default is Client mode	'deploy-mode'	No
Set CPU core and memory usage in executors	`--executor-cores` and `--executor-memory`	No
See available options by cluster manager	`./bin/spark-submit --help`	N/A

'spark-submit' Application Files

Final arguments depend on application language:

Java or Scala

<application-jar-path> <application-args>

Specifies the location of the JAR with your application and dependencies, followed by any arguments specific to the application

Python

<application-py-script> <application-args>

Specifies the application python script followed by any arguments specific to the application. Add files with '.py', '.egg' or '.zip' using the '--py-files' argument

'spark-submit' Examples

1. Launch Scala SparkPi using a jar, with master YARN. Estimate Pi with 1000 samples

```
# Launching Scala SparkPi to a YARN cluster

./bin/spark-submit \
  --class org.apache.spark.examples.SparkPi \
  --master YARN \
  /path/to/examples.jar \
  1000
```

2. Launch Python SparkPi to a Spark Standalone cluster. Estimate Pi with 1000 samples

```
# Launching Python SparkPi to a standalone cluster with master at 207.184.161.138

./bin/spark-submit \
  --master spark://207.184.161.138:7077 \
  examples/src/main/python/pi.py \
  1000
```

Application Dependencies

To manage Spark dependencies:

- Bundle projects or libraries with application so they are accessible to driver and executor processes
- For Java or Scala-based programs, create an uber-JAR with application and dependencies together so it is easy to distribute to the cluster

Python Application Dependencies

To manage Python application dependencies, ensure:

- Cluster nodes can access the required dependencies with same version
- Same Python version is used
- You use the `--py-files` argument so that `.py`, `.zip` or `.egg` files can be distributed to the cluster

```
# Launching a PySpark application with dependency on "my_python_package"

$ ./bin/spark-submit <config and options> \
    --py-files my_python_package.zip \
    my_pyspark_application.py
```

Spark Shell

Spark Shell:

- Simple way to learn Spark API
- Powerful tool to analyze data interactively
- Use in local mode or with a cluster, same options as 'spark-submit'
- Can initiate in Scala ('bin/spark-shell') and Python ('bin/pyspark')

Scala

Run <bin/spark-shell>

Python

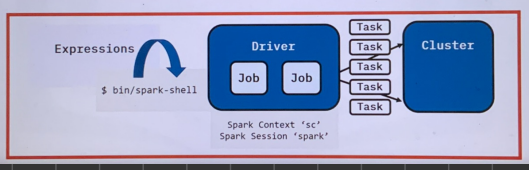
Run <bin/pyspark>

Spark Shell Environment

For both Scala and Python shells:

- SparkContext is automatically initialized and is available as 'sc'
- SparkSession is automatically available as 'spark'
- Expressions are entered into the shell and then evaluated in the driver to become jobs that are scheduled as tasks for the cluster

can start working to data intermediately



Starting Spark Shell – Local Mode

Example: Start Scala Spark-Shell and run local mode by default

```
$ bin/spark-shell
Using Spark's default log4j profile: org/apache/spark/log4j-defaults.properties
Setting default log level to "WARN".
To adjust logging level use sc.setLogLevel(newLevel). For SparkR, use setLogLevel(newLevel).
Spark context Web UI available at http://192.168.1.33:4040
Spark context available as 'sc' (master = local[*], app id = local-1614631042181).
Spark session available as 'spark'.
Welcome to
      ____              __
     / __/__  ___ _____/ /__
    _\ \/ _ \/ _ `/ __/  '_/
   /___/ .__/\_,_/_/ /_/\_\   version 3.0.1
      /_/

Using Scala version 2.12.10 (OpenJDK 64-Bit Server VM, Java 1.8.0_282)
Type in expressions to have them evaluated.
Type :help for more information.
scala>
```

Spark Shell Information Provided

- Spark's load log in the log4j-defaults.properties files
- Spark's Web UI address to view information about jobs running in the shell
- Spark Context and Spark Session variables
- Versions of important libraries in use, for example JDK and Scala

```
$ bin/spark-shell
Using Spark's default log4j profile: org/apache/spark/log4j-defaults.properties
Setting default log level to "WARN".
To adjust logging level use sc.setLogLevel(newLevel). For SparkR, use setLogLevel(newLevel).
Spark context Web UI available at http://192.168.1.33:4040
Spark context available as 'sc' (master = local[*], app id = local-1614631042181).
Spark session available as 'spark'.
Welcome to
      ____              __
     / __/__  ___ _____/ /__
    _\ \/ _ \/ _ `/ __/  '_/
   /___/ .__/\_,_/_/ /_/\_\   version 3.0.1
      /_/

Using Scala version 2.12.10 (OpenJDK 64-Bit Server VM, Java 1.8.0_282)
Type in expressions to have them evaluated.
Type :help for more information.
scala>
```

Spark Shell Example – Run Code

```
scala> val df = spark.range(10)
df: org.apache.spark.sql.Dataset[Long] = [id: bigint]

scala> df.withColumn("mod", expr("id % 2")).show(4)
+---+---+
| id|mod|
+---+---+
|  0|  0|
|  1|  1|
|  2|  0|
|  3|  1|
+---+---+
```

1. Launch Scala Spark Shell
2. Create distributed DataFrame with column 'id' and 10 values (0-9)
3. Add a column that evaluates an SQL expression for modulo of 2 and show first four (4) results