# An Introduction to **Apache Spark**

Zebula Sampedro
*sampedro@colorado.edu*

Basics → RDDs → Architecture → Spark on Janus

**Basics** → RDDs → Architecture → Spark on Janus

# What is Spark?

- A general-purpose engine for processing huge data.
- Exposes APIs in Java, Scala, Python, and R.
- Base project for a number of special-focus libraries
    - MLLib - *spark.apache.org/mllib/*
    - SparkSQL - *spark.apache.org/sql/*
    - SparkStreaming - *spark.apache.org/streaming/*
    - GraphX - *spark.apache.org/graphx/*

# Spark vs. Hadoop

- Spark doesn’t replace the entire Hadoop project.
- Hadoop consists of three primary projects:
    - HDFS (Distributed filesystem)
    - Yarn (Resource manager)
    - MapReduce (Programming model/implementation)

# Spark vs. Hadoop

- Spark doesn’t replace the entire Hadoop project.
- Hadoop consists of three primary projects:
  - HDFS
  - Yarn
  - **MapReduce**

**Spark is a potential replacement for MapReduce**

# Core goals of Spark

Ad-hoc queries, interactive data

Scalable support for iterative workflows

# Core goals of Spark

Ad-hoc queries, interactive data

Scalable support for iterative workflows

**Spark accomplishes these goals with a data structure called Resilient Distributed Datasets (RDDs) that allow data to be persisted in-memory.**

Basics → **RDDs** → Architecture → Spark on Janus

# RDDs

Resilient Distributed Datasets (RDDs) are the core data structure in Spark. They are designed to be configurable, parallel, and fault-tolerant:

Configurable

- Users can persist intermediate results in memory.
- Users can, to a limited degree, control data placement.
- Data can be placed in-memory, on disk, or a combination of both.

# RDDs

Resilient Distributed Datasets (RDDs) are the core data structure in Spark. They are designed to be configurable, parallel, and fault-tolerant:

Parallel

- RDDs are divided into partitions
- User can explicitly control partition count

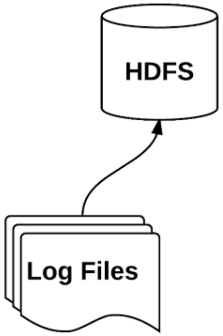
# RDDs

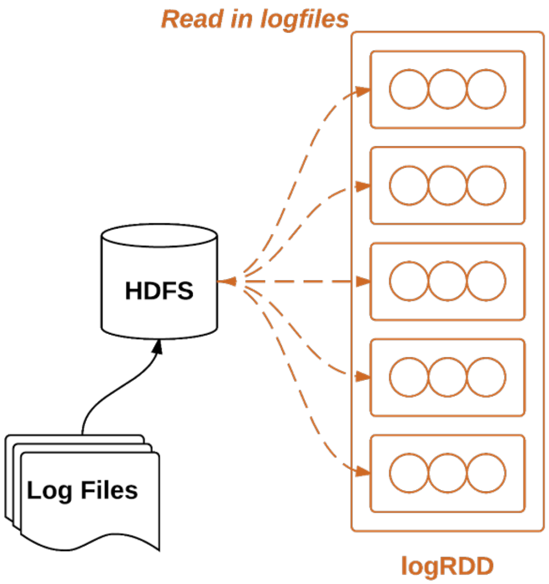
Resilient Distributed Datasets (RDDs) are the core data structure in Spark. They are designed to be configurable, parallel, and fault-tolerant:

Fault-tolerant

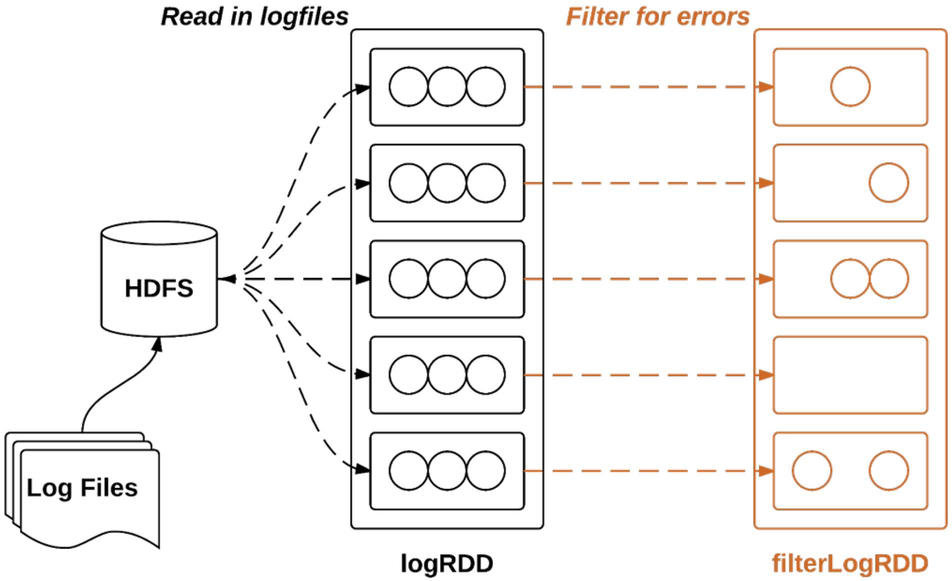
- Solving fault tolerance with replication scales poorly.
- RDDs don’t replicate, they trace partition lineage with a DAG.
- Evacuated or lost partitions can be recomputed efficiently
- Lazily-evaluated

# Building the DAG

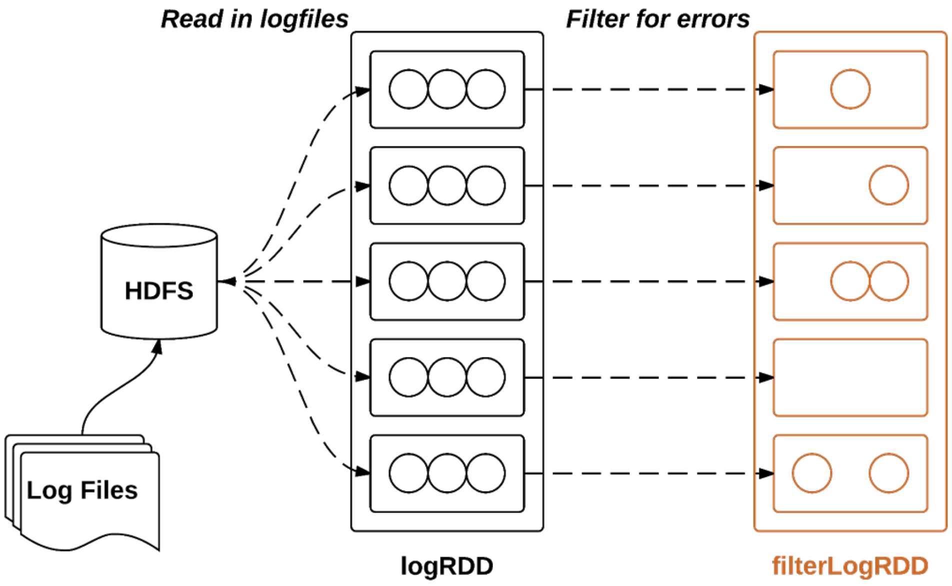




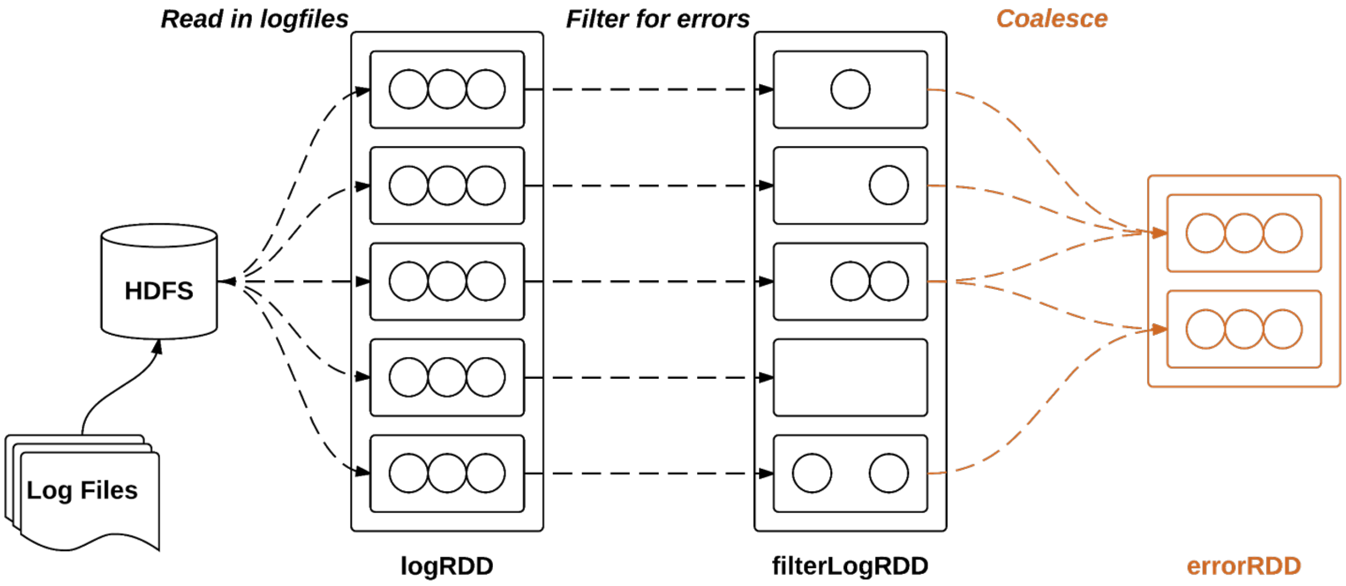
```
> logRDD = sc.textFile('/logs/*.csv', 5)
```



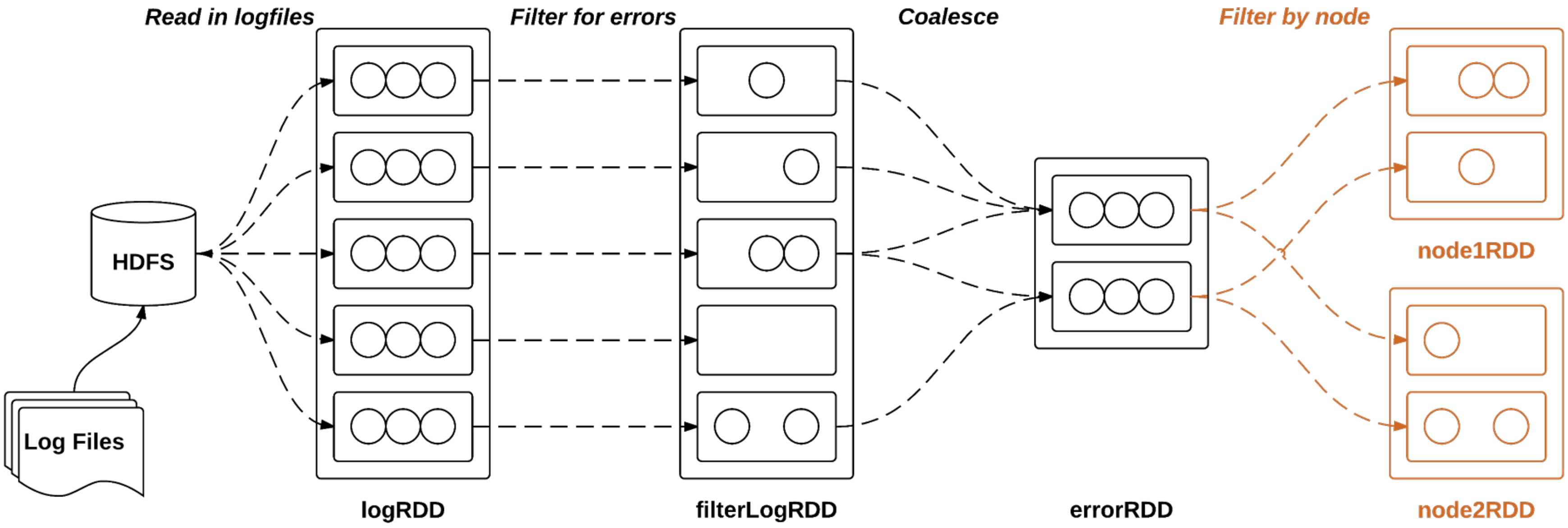
```
> logRDD = sc.textFile('/logs/*.csv', 5)

> filterLogRDD = logRDD.filter(lambda line: 'error' in line)
```



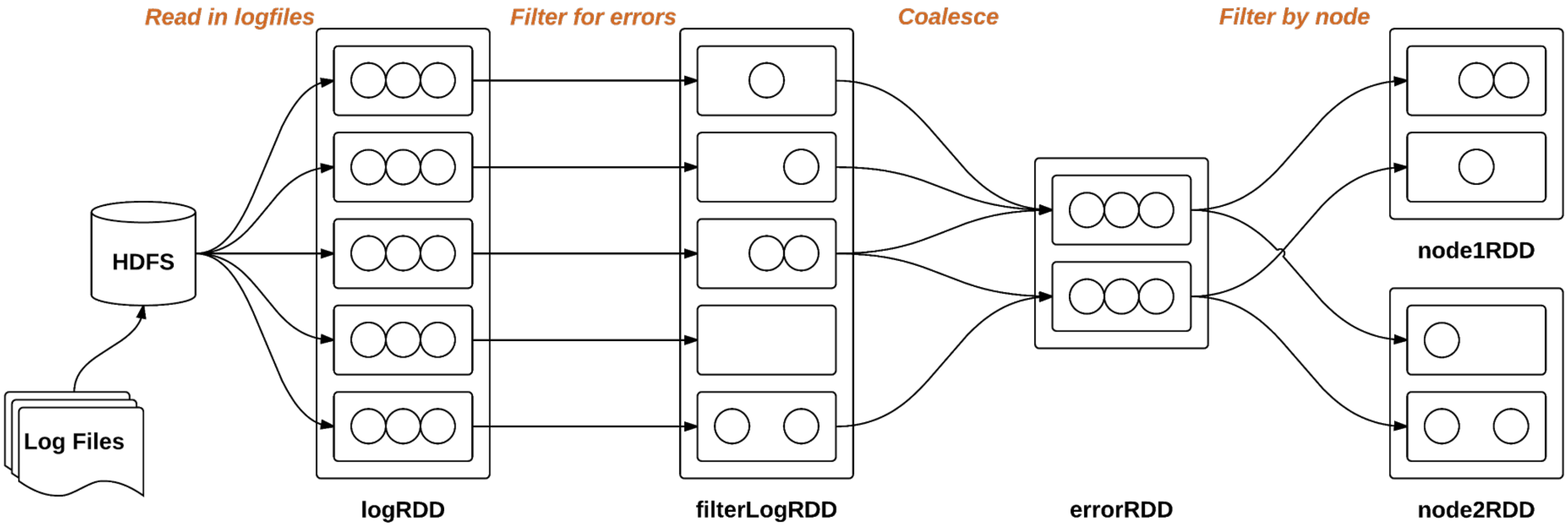
```
> logRDD = sc.textFile('/logs/*.csv', 5)

> filterLogRDD = logRDD.filter(lambda line: 'error' in line)

> filterLogRDD.cache()
```



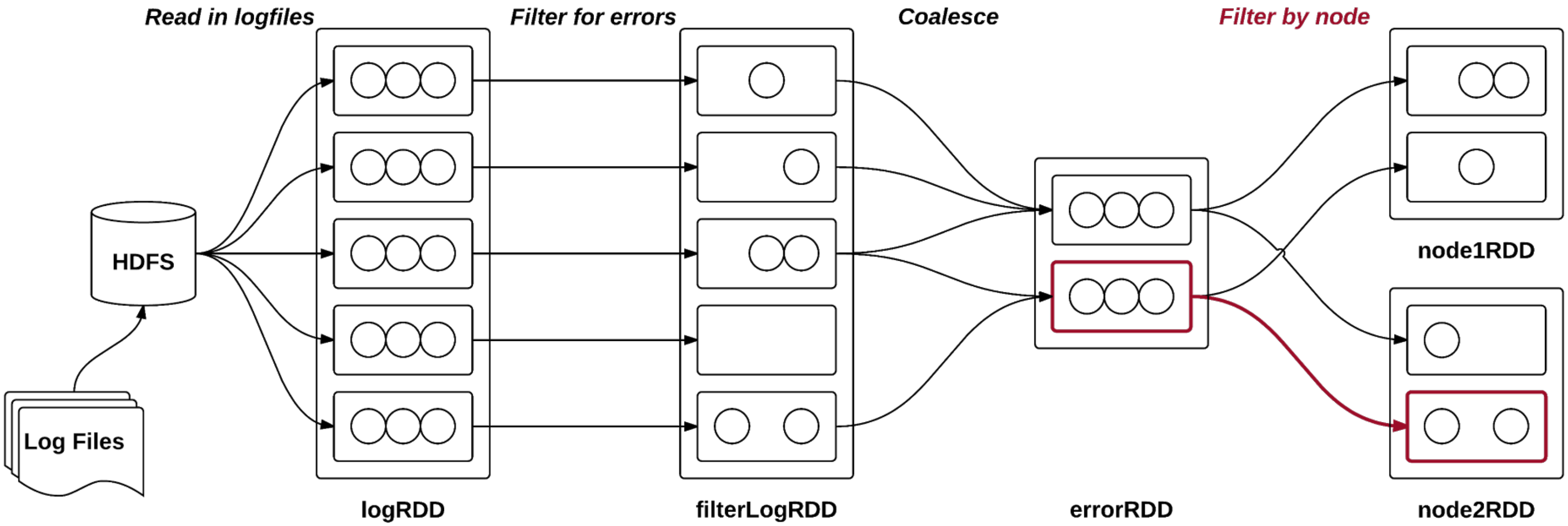
```
> logRDD = sc.textFile('/logs/*.csv', 5)

> filterLogRDD = logRDD.filter(lambda line: 'error' in line)

> filterLogRDD.cache()

> errorRDD = filterLogRDD.coalesce(2)
```



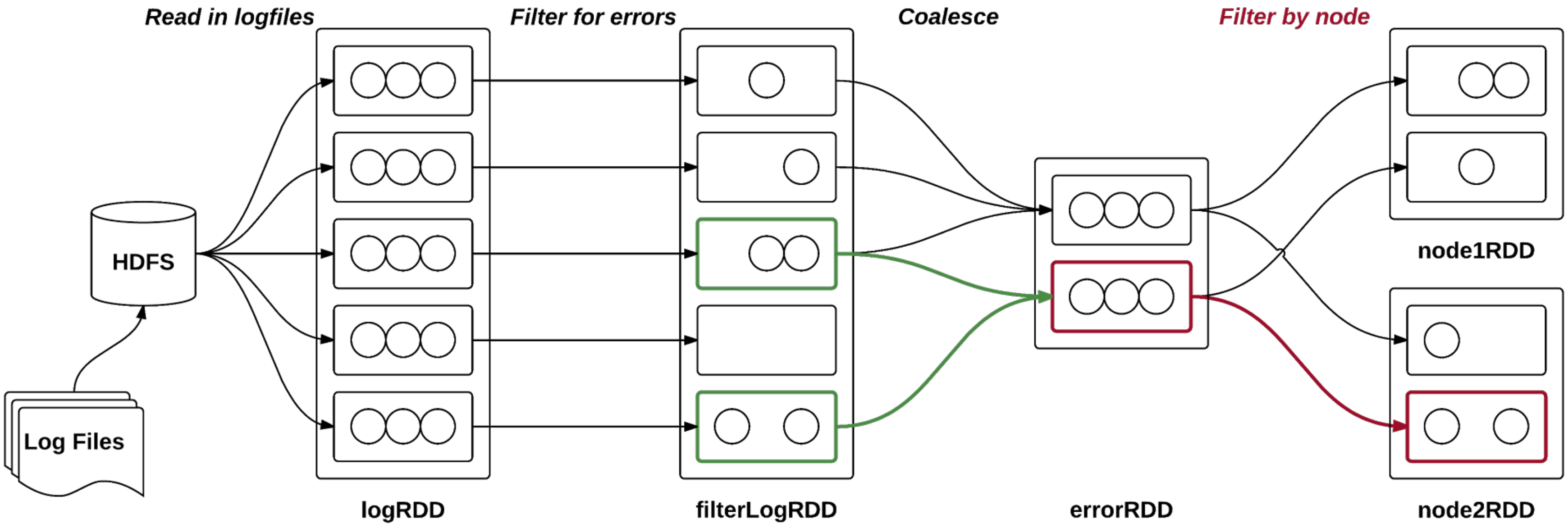
```
> logRDD = sc.textFile('/logs/*.csv', 5)

> filterLogRDD = logRDD.filter(lambda line: 'error' in line)

> filterLogRDD.cache()

> errorRDD = filterLogRDD.coalesce(2)

> node1RDD = errorRDD.filter(lambda line: 'node1' in line)

> node2RDD = errorRDD.filter(lambda line: 'node2' in line)
```



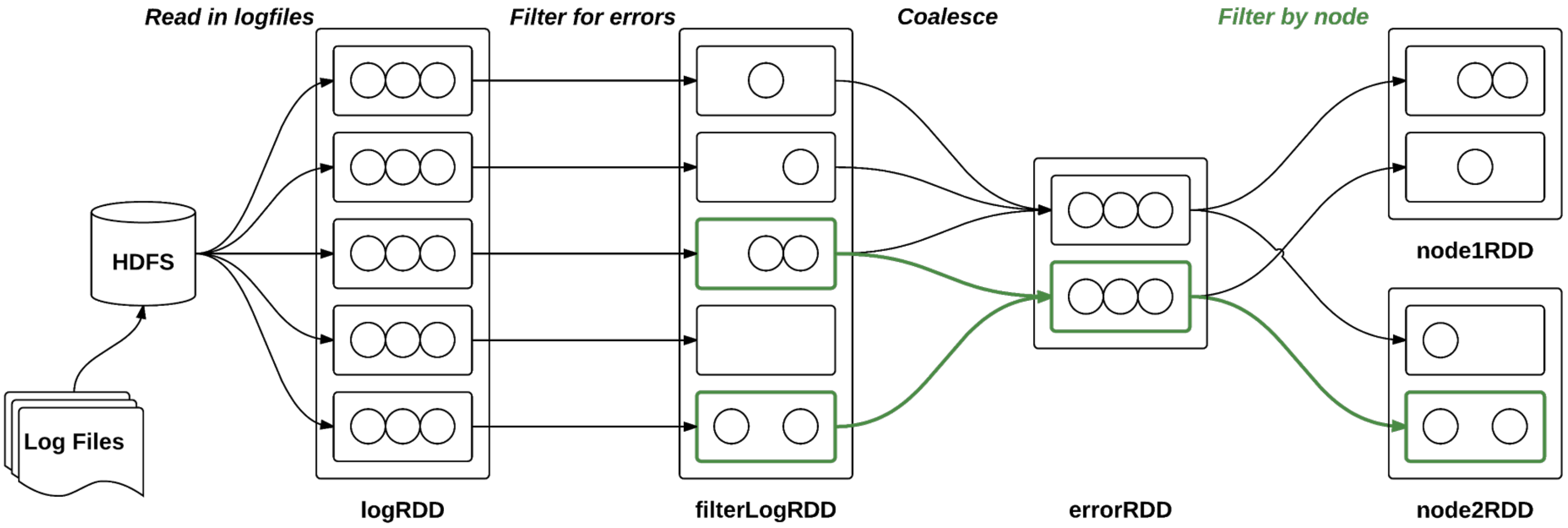
```
> logRDD = sc.textFile('/logs/*.csv', 5)

> filterLogRDD = logRDD.filter(lambda line: 'error' in line)

> filterLogRDD.cache()

> errorRDD = filterLogRDD.coalesce(2)

> node1RDD = errorRDD.filter(lambda line: 'node1' in line)

> node2RDD = errorRDD.filter(lambda line: 'node2' in line)

> node1RDD.collect()
```



```
> logRDD = sc.textFile('/logs/*.csv', 5)

> filterLogRDD = logRDD.filter(lambda line: 'error' in line)

> filterLogRDD.cache()

> errorRDD = filterLogRDD.coalesce(2)

> node1RDD = errorRDD.filter(lambda line: 'node1' in line)

> node2RDD = errorRDD.filter(lambda line: 'node2' in line)

> node1RDD.collect()
```



```
> logRDD = sc.textFile('/logs/*.csv', 5)

> filterLogRDD = logRDD.filter(lambda line: 'error' in line)

> filterLogRDD.cache()

> errorRDD = filterLogRDD.coalesce(2)

> node1RDD = errorRDD.filter(lambda line: 'node1' in line)

> node2RDD = errorRDD.filter(lambda line: 'node2' in line)

> node1RDD.collect()
```



```
> logRDD = sc.textFile('/logs/*.csv', 5)

> filterLogRDD = logRDD.filter(lambda line: 'error' in line)

> filterLogRDD.cache()

> errorRDD = filterLogRDD.coalesce(2)

> node1RDD = errorRDD.filter(lambda line: 'node1' in line)

> node2RDD = errorRDD.filter(lambda line: 'node2' in line)

> node1RDD.collect()
```

Great resource for an in-depth explanation of RDDs:
https://www.usenix.org/conference/nsdi12/technical-sessions/presentation/zaharia

Basics → RDDs → **Architecture** → Spark on Janus

# Architecture - Deploy Modes

Different deploy modes:

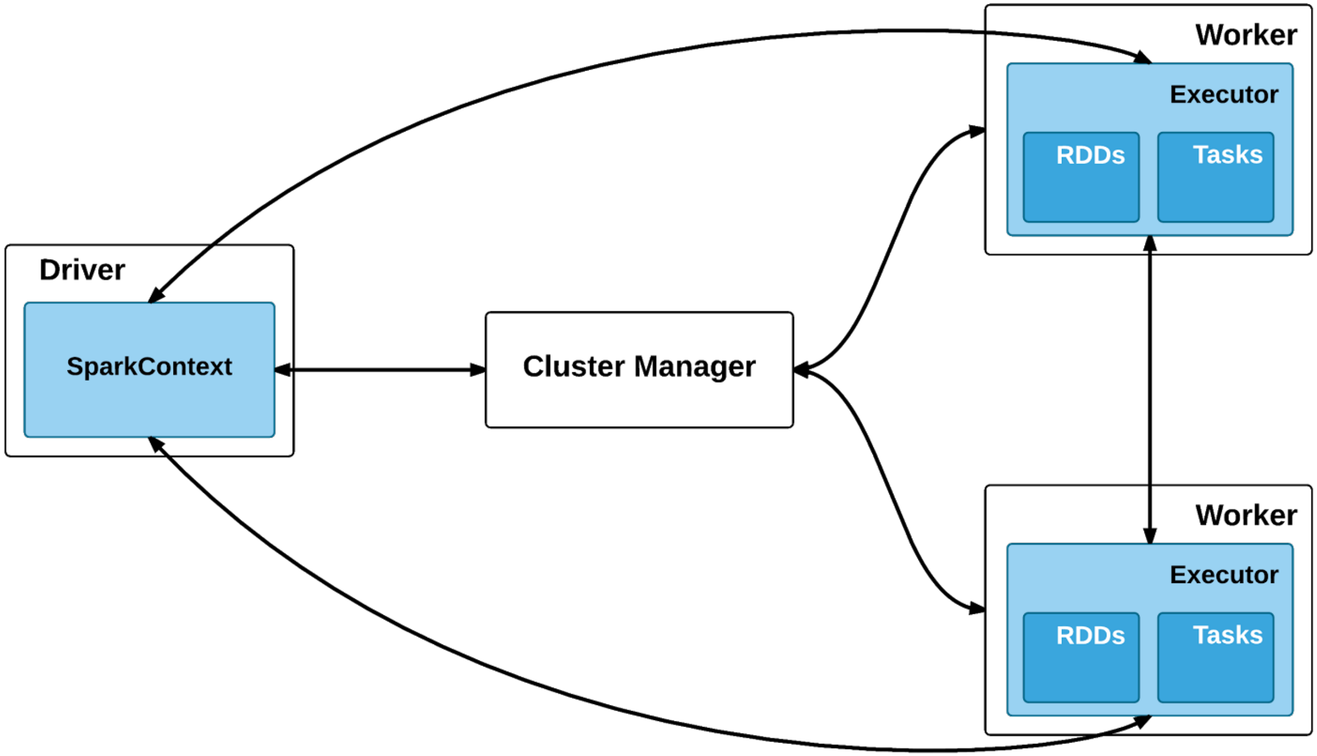
- Local
- Standalone
- Yarn
- Mesos

# Architecture - Deploy Modes

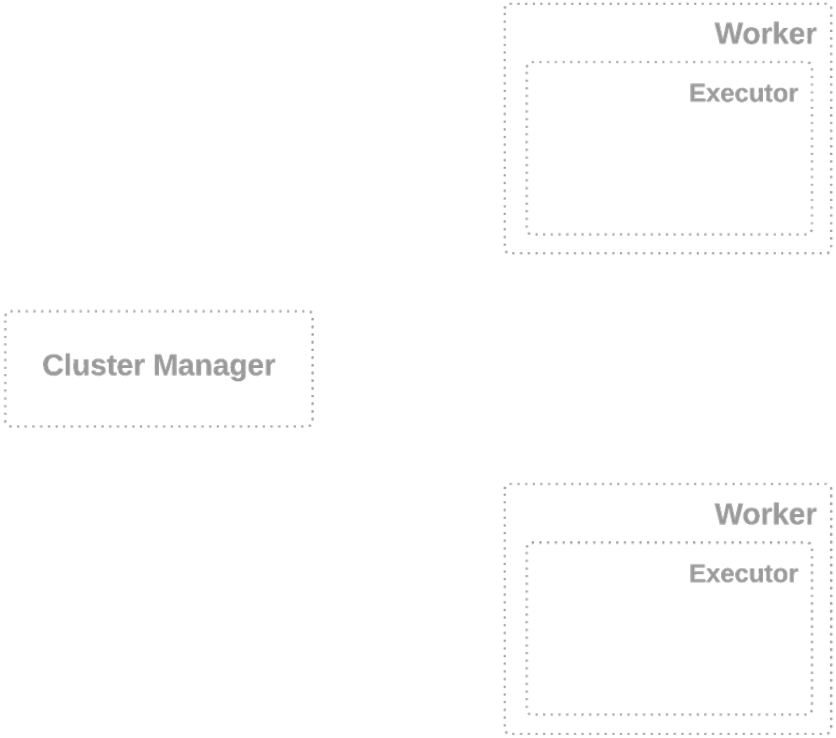
Different deploy modes:

- Local
- **Standalone**
- Yarn
- Mesos

# Architecture - Cluster Mode



```
$ source spark-env.sh
```

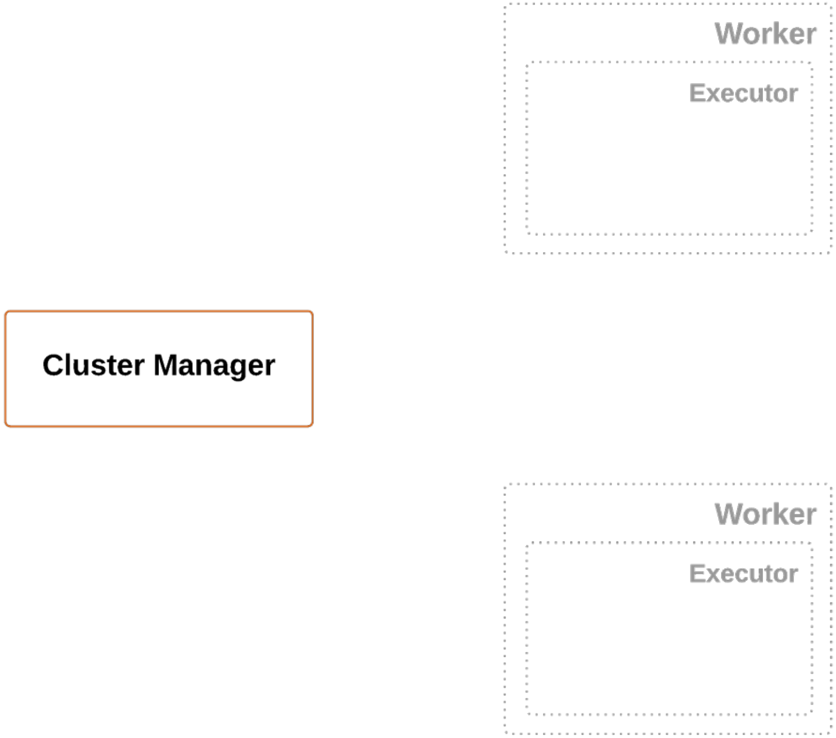


# spark-env.sh

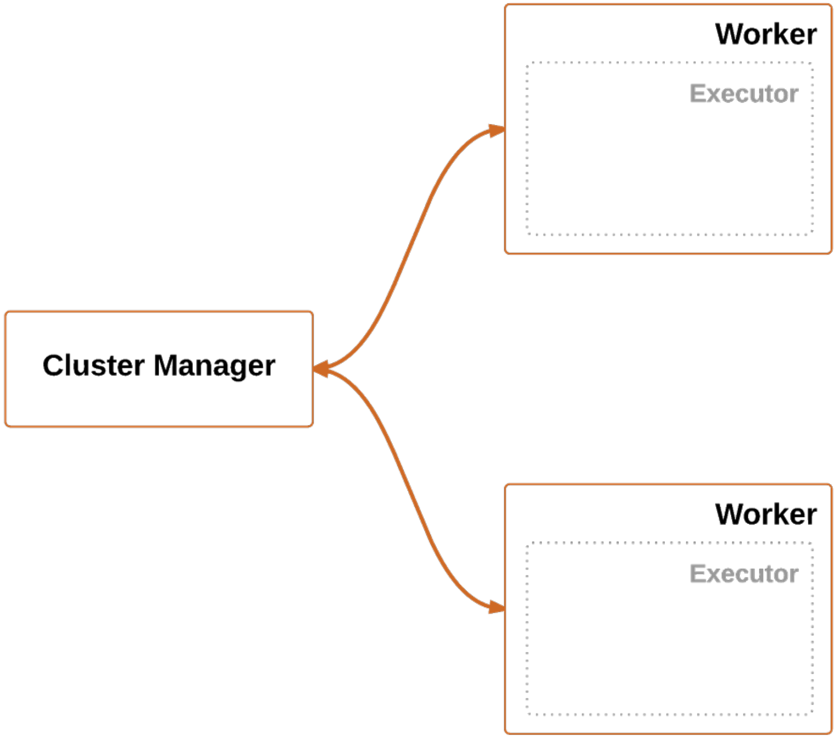
*$SPARK_HOME/conf/spark-env.sh*

**SPARK_LOCAL_DIRS** **-** Disks to use for spillover/local persistence
**SPARK_WORKER_CORES** **-** Max cores Worker can allocate to Executor JVMs
**SPARK_WORKER_MEMORY** **-** Max memory Worker can allocate to Executor JVMs
**SPARK_DAEMON_MEMORY** **-** Memory to allocate for Master and Worker JVMs

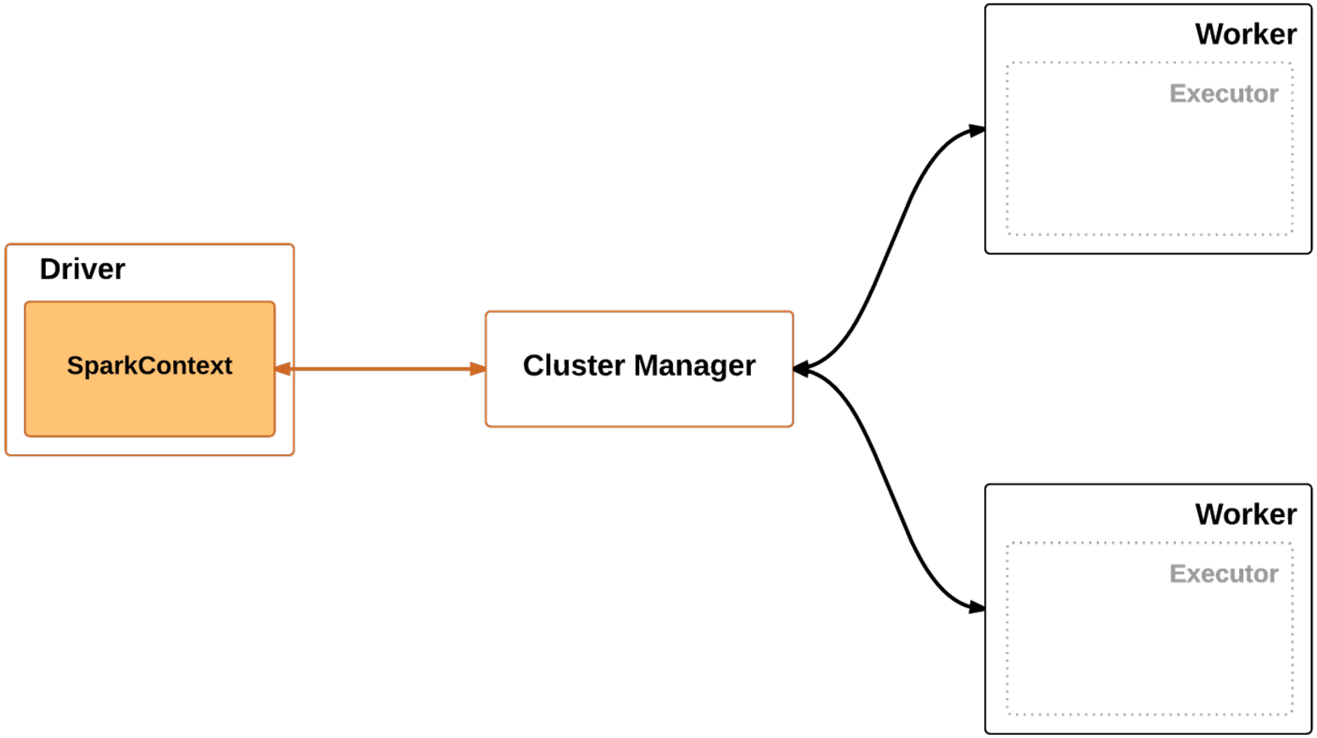
```
$ source spark-env.sh

$ ./start-master.sh
```



```
$ source spark-env.sh

$ ./start-master.sh

$ ./start-slave.sh $MASTER
```



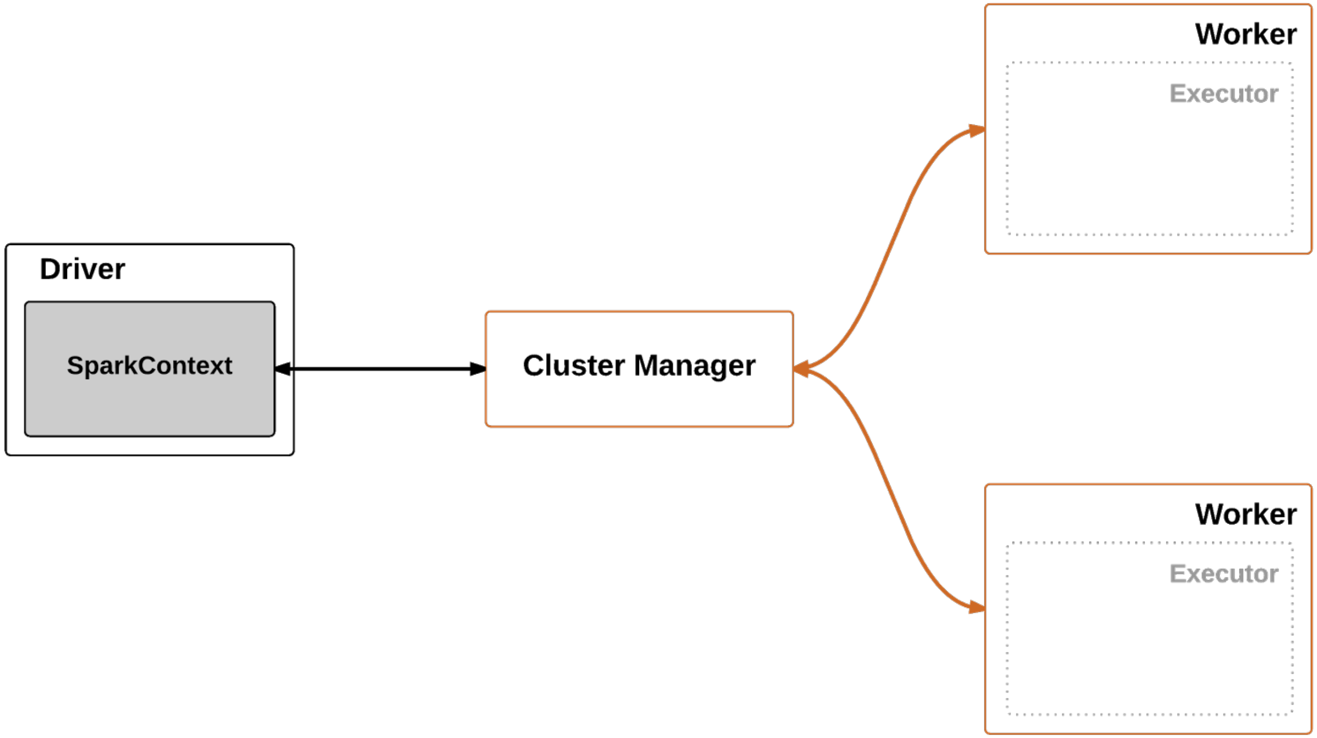
```
$ source spark-env.sh

$ ./start-master.sh

$ ./start-slave.sh $MASTER

$ ./pyspark --master $MASTER
```



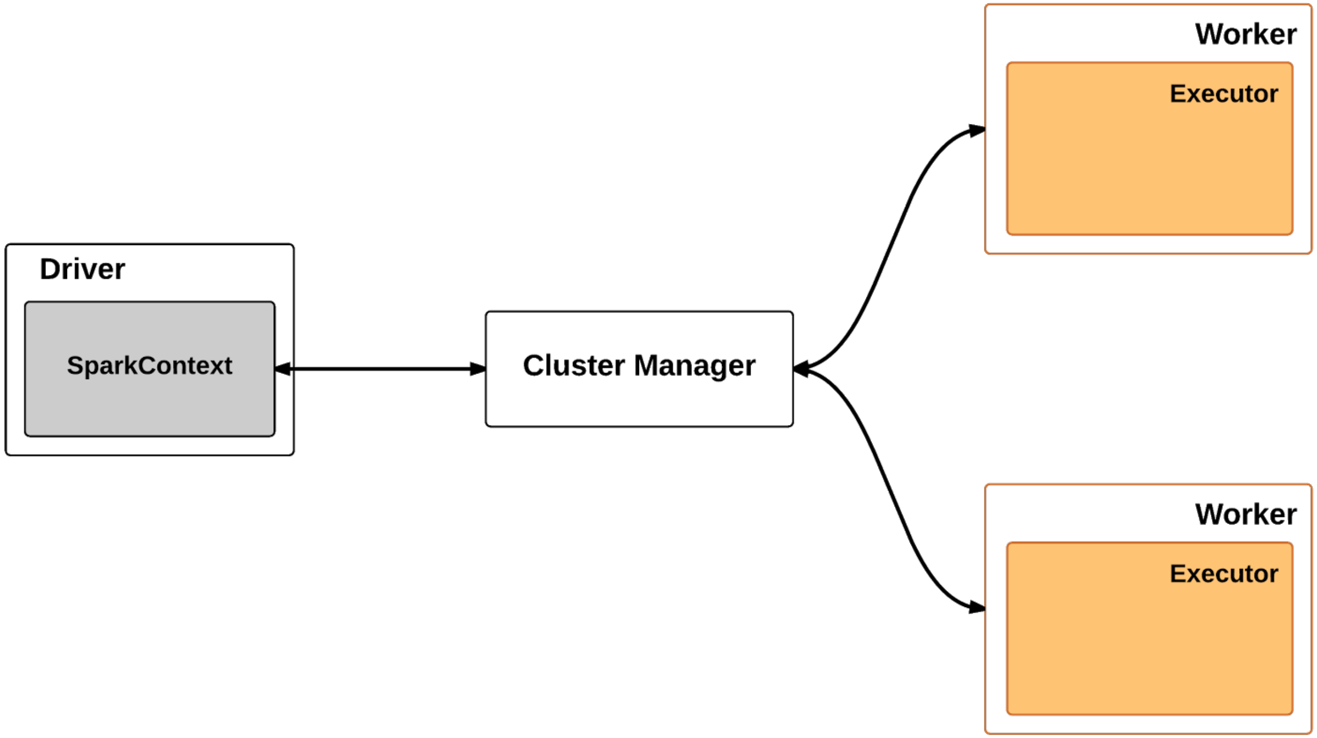
# submit options

**executor-memory** - Max memory to allocate per Executor JVM
**driver-memory** - Memory to allocate to the Driver JVM
**spark.cores.max** - In standalone, max cores to request from cluster
**spark.local.dir** - Location to use for application scratch space
**spark.driver.maxResultSize** - Maximum allowable result size sent to Driver

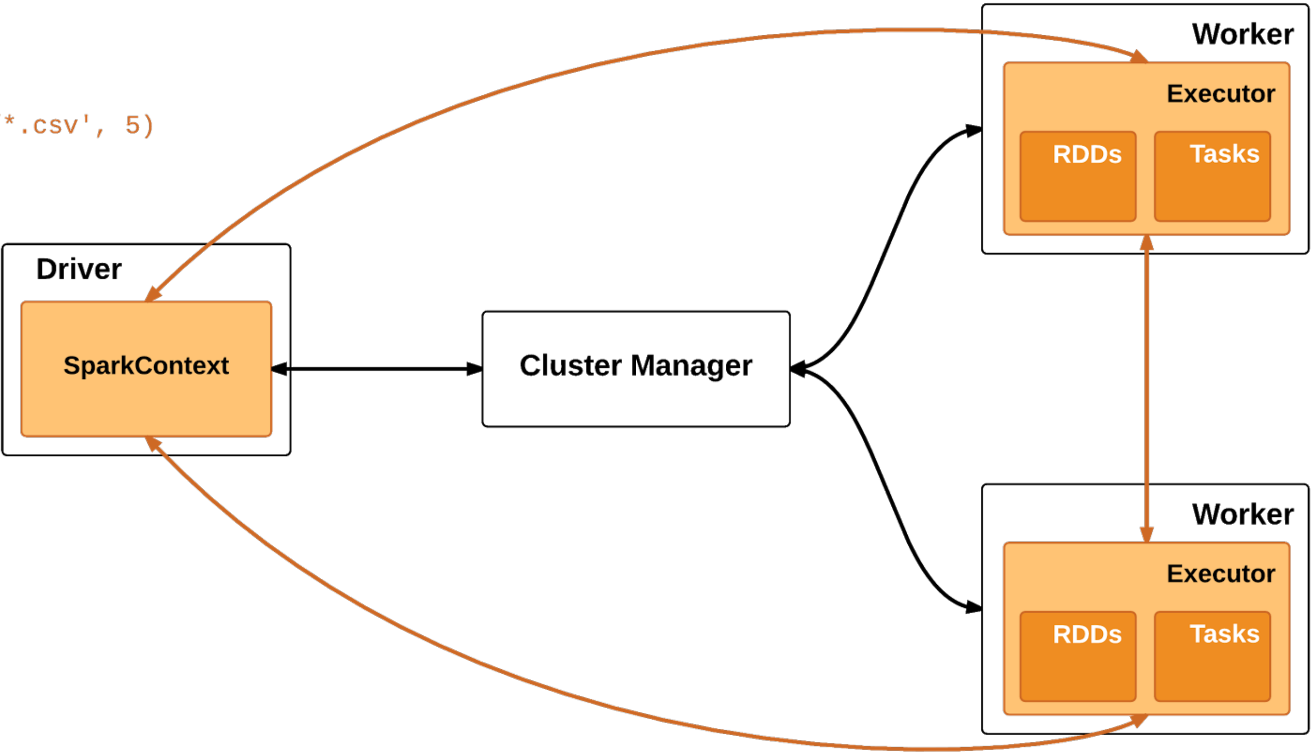
```
$ source spark-env.sh

$ ./start-master.sh

$ ./start-slave.sh $MASTER

$ ./pyspark --master $MASTER
```



```
$ source spark-env.sh

$ ./start-master.sh

$ ./start-slave.sh $MASTER

$ ./pyspark --master $MASTER
```



```
$ source spark-env.sh

$ ./start-master.sh

$ ./start-slave.sh $MASTER

$ ./pyspark --master $MASTER

> logRDD = sc.textFile('/logs/*.csv', 5)
```



Basics → RDDs → Architecture → **Spark on Janus**

# Sparkling on Janus

The basics of running Spark on Janus:

- Standalone mode
- Transient Spark clusters
- Lustre, no HDFS

# Sparkling on Janus

A good example of running a self-contained applications can be found here:

Official documentation on self-contained applications

A good example of running a self-contained applications can be found here:

Official documentation on self-contained applications

**We will be running interactive jobs in PySpark shell**

# Sparkling on Janus

First, we need to login to Janus, clone the repo, and start an interactive job:

```
$ ssh <username>@tutorial-login.rc.colorado.edu
$ git clone https://github.com/ResearchComputing/RMACC2015-Spark.git
$ cd RMACC2015-Spark/spark-setup-scripts
$ ml slurm
$ salloc --nodes=2 -t 01:30:00 -A crctutorial --reservation=rmacc-tutorials
```

At this point, you are ready to use the *spark-cluster.sh* script provided and open the PySpark shell:

```
$ source spark-cluster.sh start
$ $SPARK_HOME/bin/pyspark --master=$MASTER --driver-memory=12g
```

## *An explanation of spark-cluster.sh*

Set environment variables for Spark:

```
export SPARK_HOME=/projects/$USER/spark-1.4.1-bin-hadoop2.6
export SPARK_CONF_DIR=$SPARK_HOME/conf
export SPARK_HOSTFILE=$SPARK_CONF_DIR/spark_hostfile
export CLASSPATH=$SPARK_HOME/lib/spark-examples-1.4.1-hadoop2.6.0.jar
```

Then, remove any configuration from previous runs:

```
rm $SPARK_HOSTFILE $SPARK_HOME/conf/slaves
```

## Sparkling on Janus

Generate configuration for currently allocated nodes:

```
srun hostname >> $SPARK_HOSTFILE


sed -i 's/$/ib/' $SPARK_HOSTFILE
tail -n +2 $SPARK_HOSTFILE | sort -u >> $SPARK_CONF_DIR/slaves


export SPARK_MASTER_IP=$(sort -u $SPARK_HOSTFILE | head -n 1)
export MASTER=spark://$SPARK_MASTER_IP:7077
```

Then, copy spark-env.sh to the conf dir and source it:

```
cp spark-env.sh $SPARK_CONF_DIR/spark-env.sh
source $SPARK_CONF_DIR/spark-env.sh
```

Set appropriate commands for starting (or stopping) master and slaves:

```
if [ "$1" == "start" ]; then
   cmd_master="$SPARK_HOME/sbin/start-master.sh"
   cmd_slave="$SPARK_HOME/sbin/spark-daemon.sh --config $SPARK_CONF_DIR start
org.apache.spark.deploy.worker.Worker 1 $MASTER"
elif [ "$1" == "stop" ]; then
   cmd_master="$SPARK_HOME/sbin/stop-master.sh"
   cmd_slave="$SPARK_HOME/sbin/spark-daemon.sh --config $SPARK_CONF_DIR stop org.
apache.spark.deploy.worker.Worker 1"
else
   exit 1
fi
```

# Sparkling on Janus

Finally, run the master and slave commands across the node pool:

```
$cmd_master


for slave in $(sort -u $SPARK_CONF_DIR/slaves)
   do
       ssh $slave "$cmd_slave"
   done
```