An Introduction to **Apache Spark**

February Meetup

Daniel Milroy

daniel.milroy@colorado.edu

Slides courtesy of Zebula Sampedro

sampedro@colorado.edu

Basics ➔ RDDs ➔ Architecture

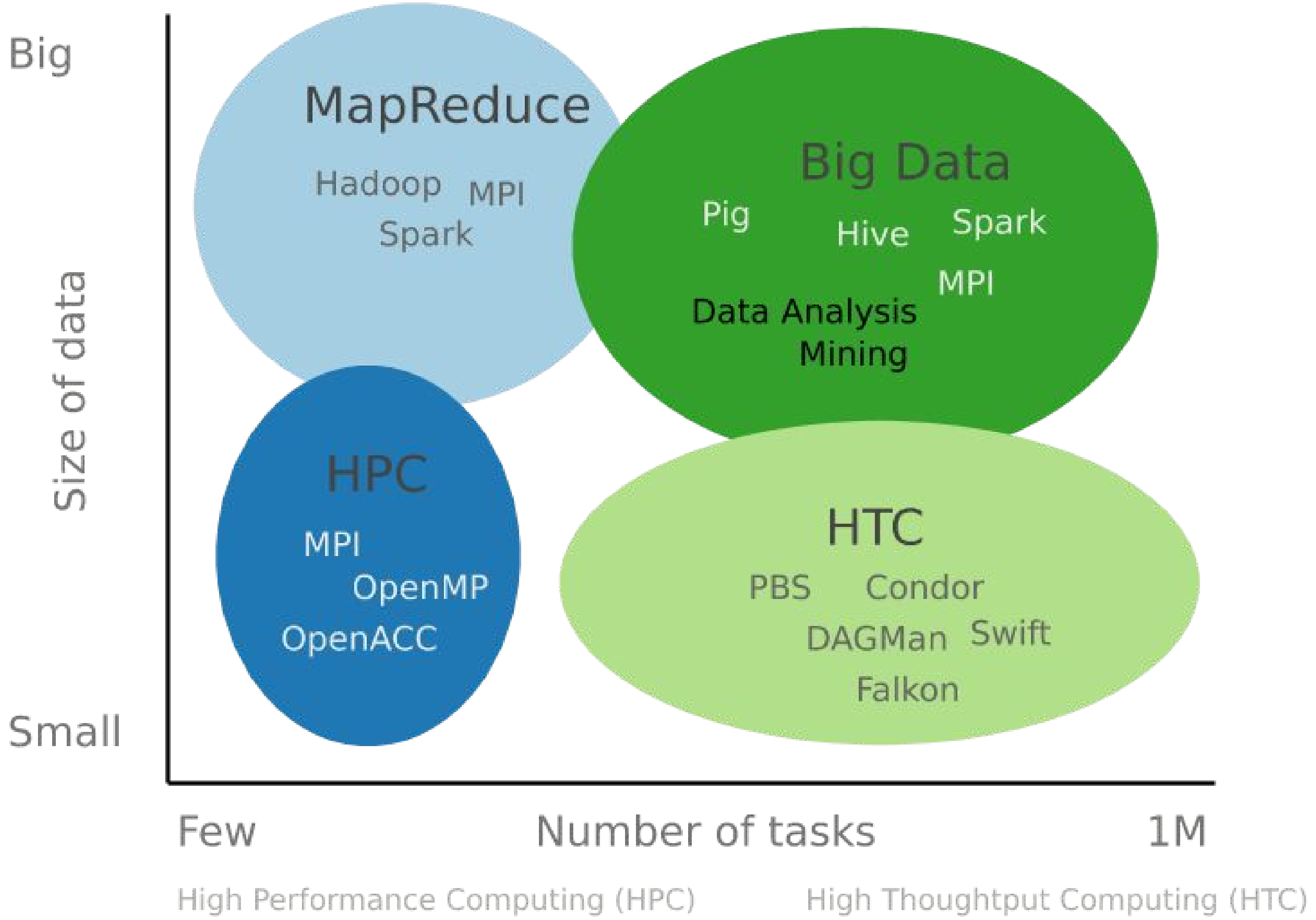
Slides, examples, and data available at:

https://github.com/ResearchComputing/Final_Tutorials/tree/master/Intro_Spark/Spring2016

Or clone the repo from:

https://github.com/milroy/Spark-Meetup

Basics → RDDs → Architecture



Many-Task Computing: Bridging the Gap between High Throughput Computing and High Performance Computing

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

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

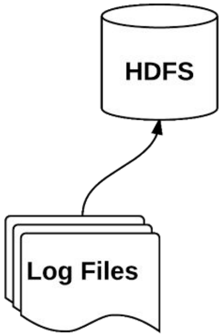
RDDs

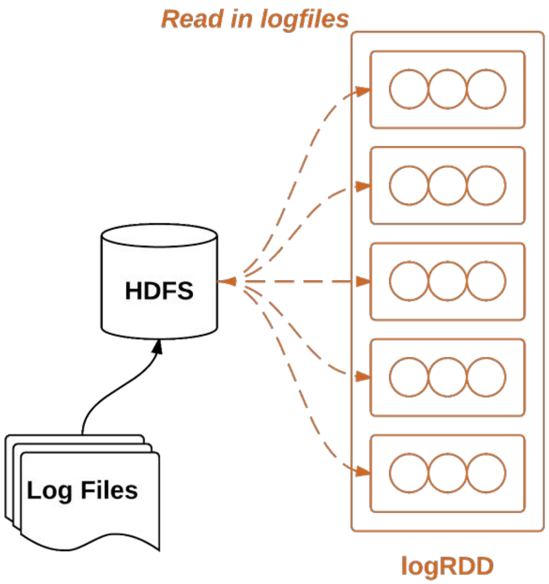
Resilient Distributed Datasets (RDDs) are the core data structure in Spark. They are designed to be configurable, parallel, and fault-tolerant:

Fault-tolerant

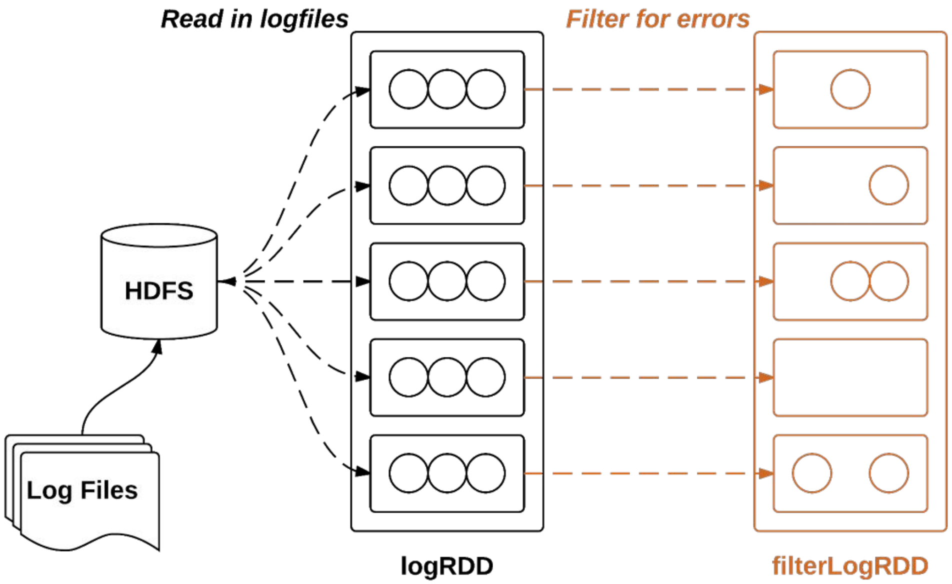
- Solving fault tolerance with replication scales poorly.
- RDDs don’t replicate, they trace partition lineage with a DAG.
- Evacuated or lost partitions can be recomputed efficiently
- Lazily-evaluated

Building the DAG

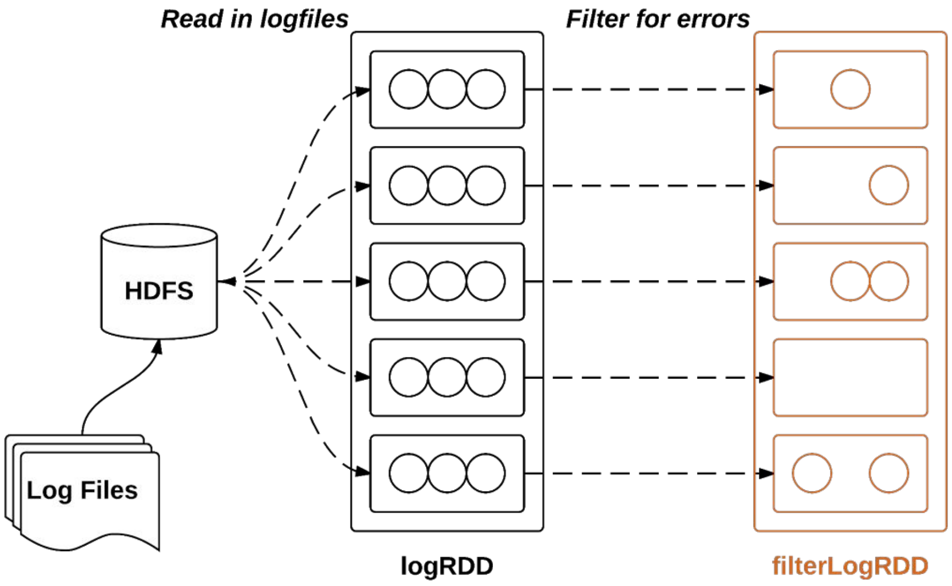




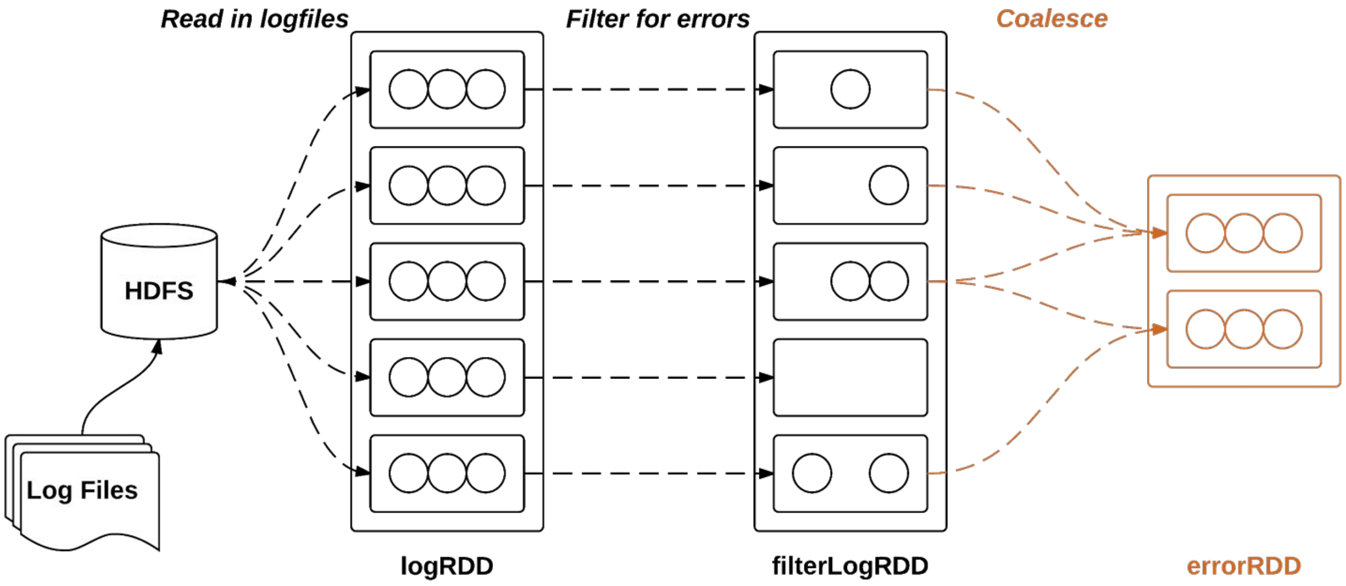
```
> logRDD = sc.textFile('/logs/*.csv', 5)
```



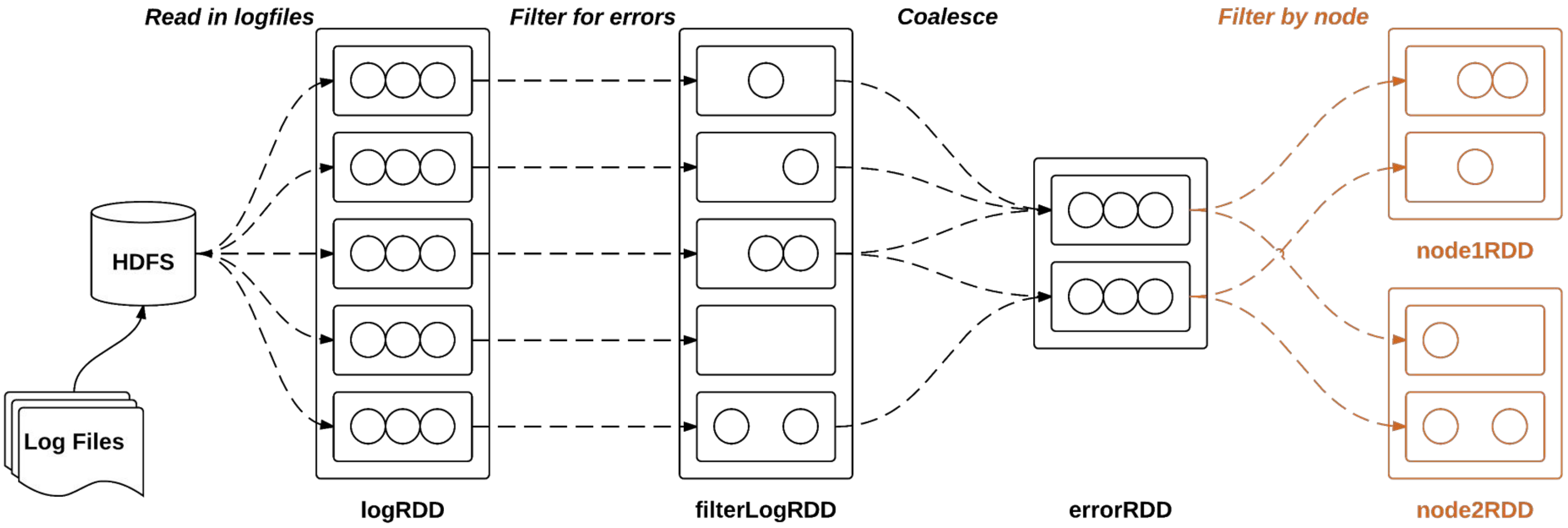
```
> logRDD = sc.textFile('/logs/*.csv', 5)

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



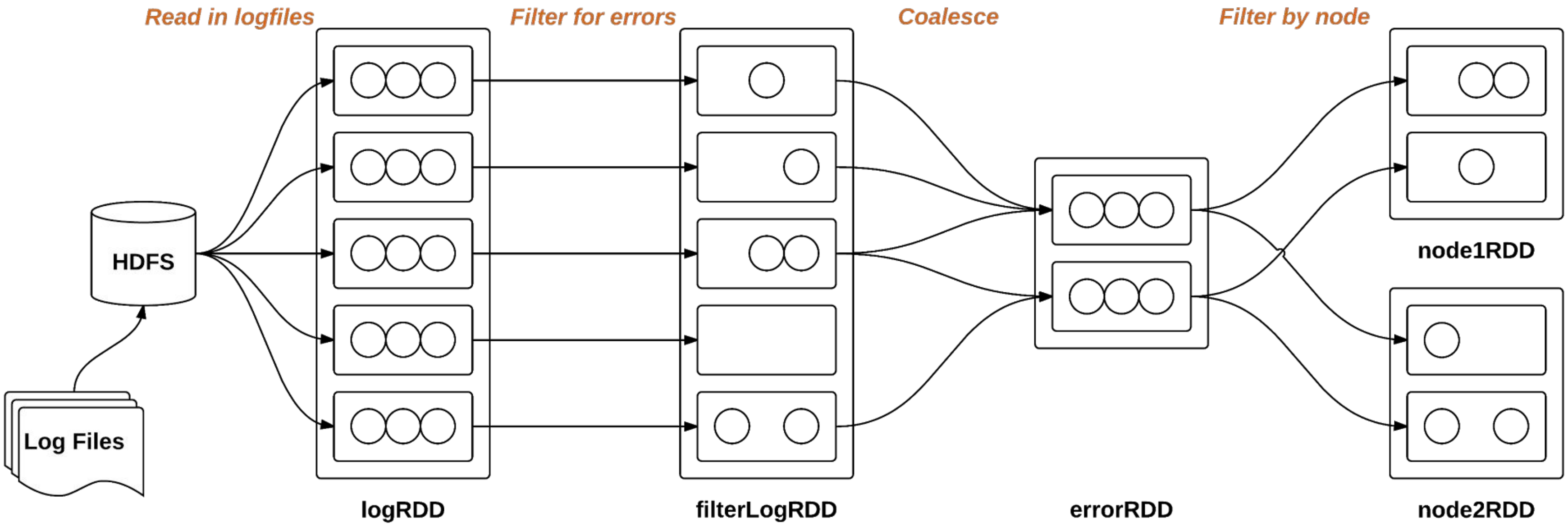
```
> logRDD = sc.textFile('/logs/*.csv', 5)

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

> filterLogRDD.cache()
```



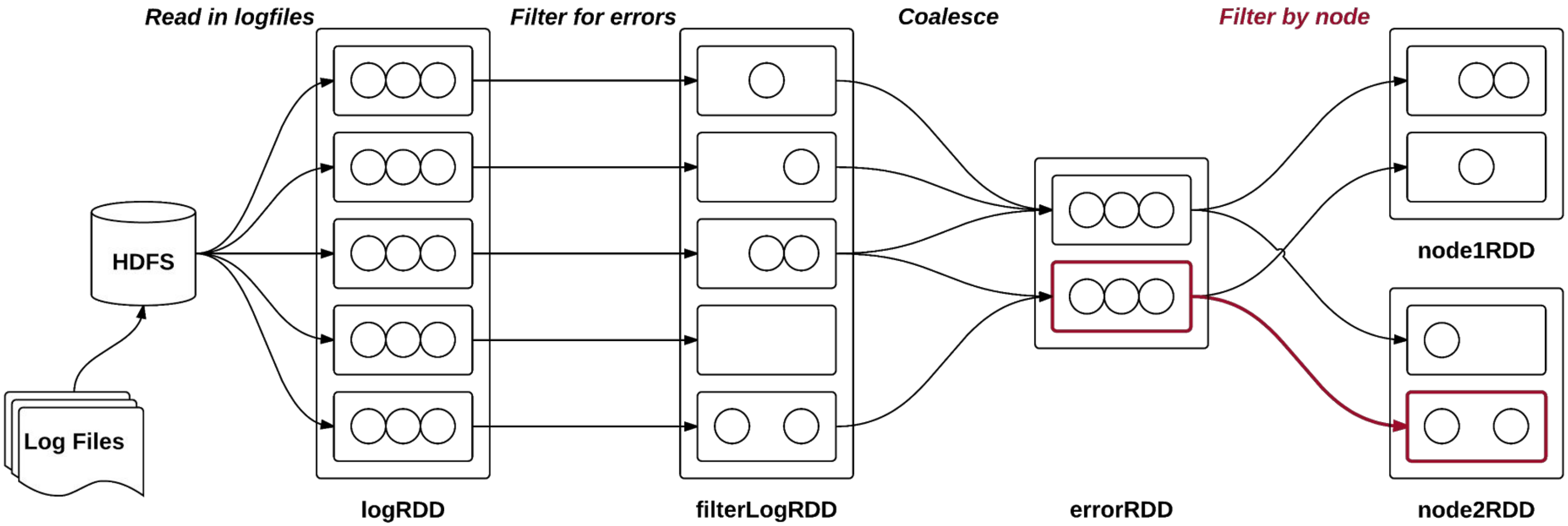
```
> logRDD = sc.textFile('/logs/*.csv', 5)

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

> filterLogRDD.cache()

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



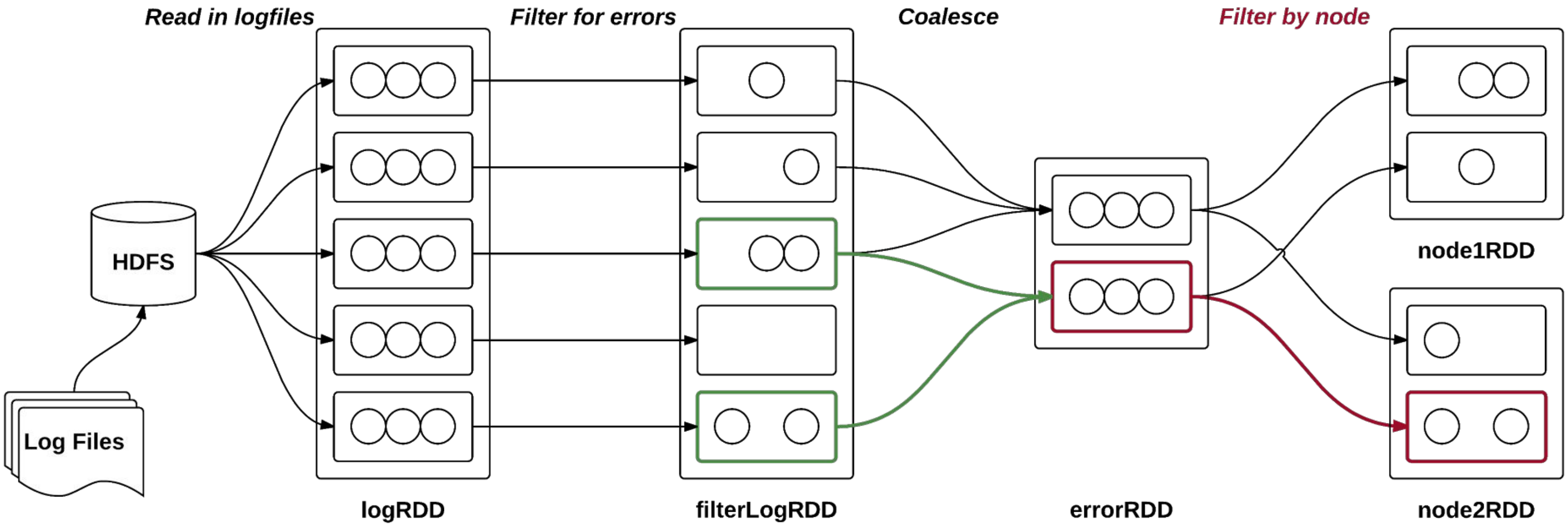
```
> logRDD = sc.textFile('/logs/*.csv', 5)

> 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)
```



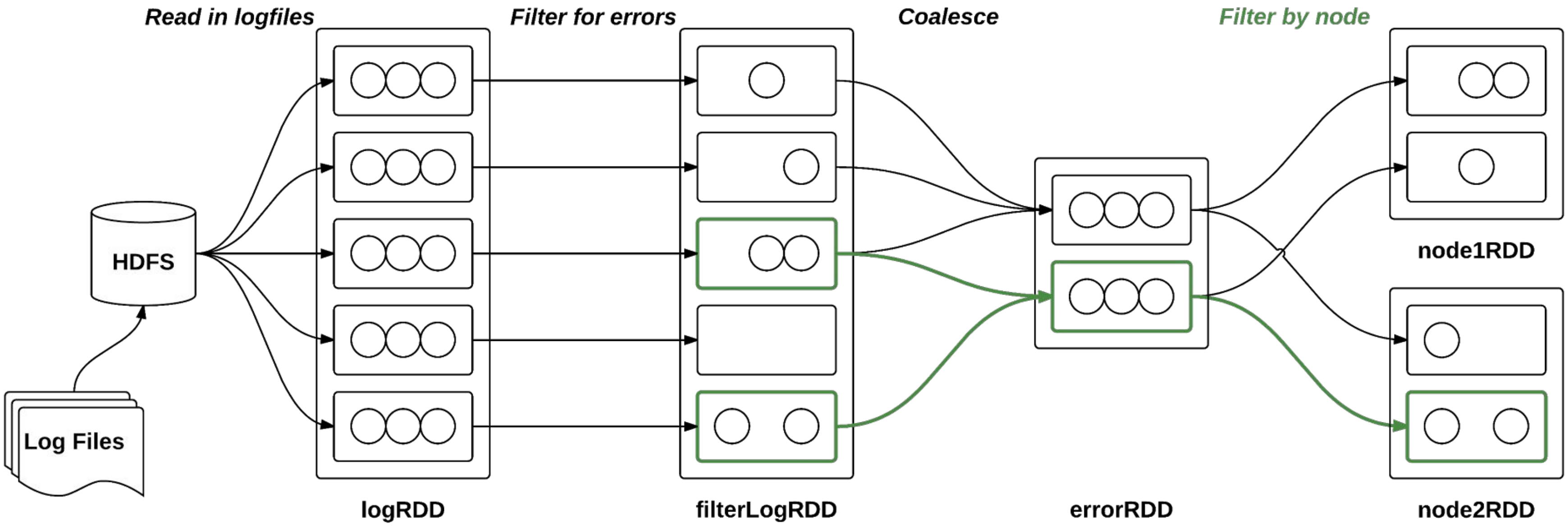
```
> logRDD = sc.textFile('/logs/*.csv', 5)

> 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**

Architecture - Deploy Modes

Different deploy modes:

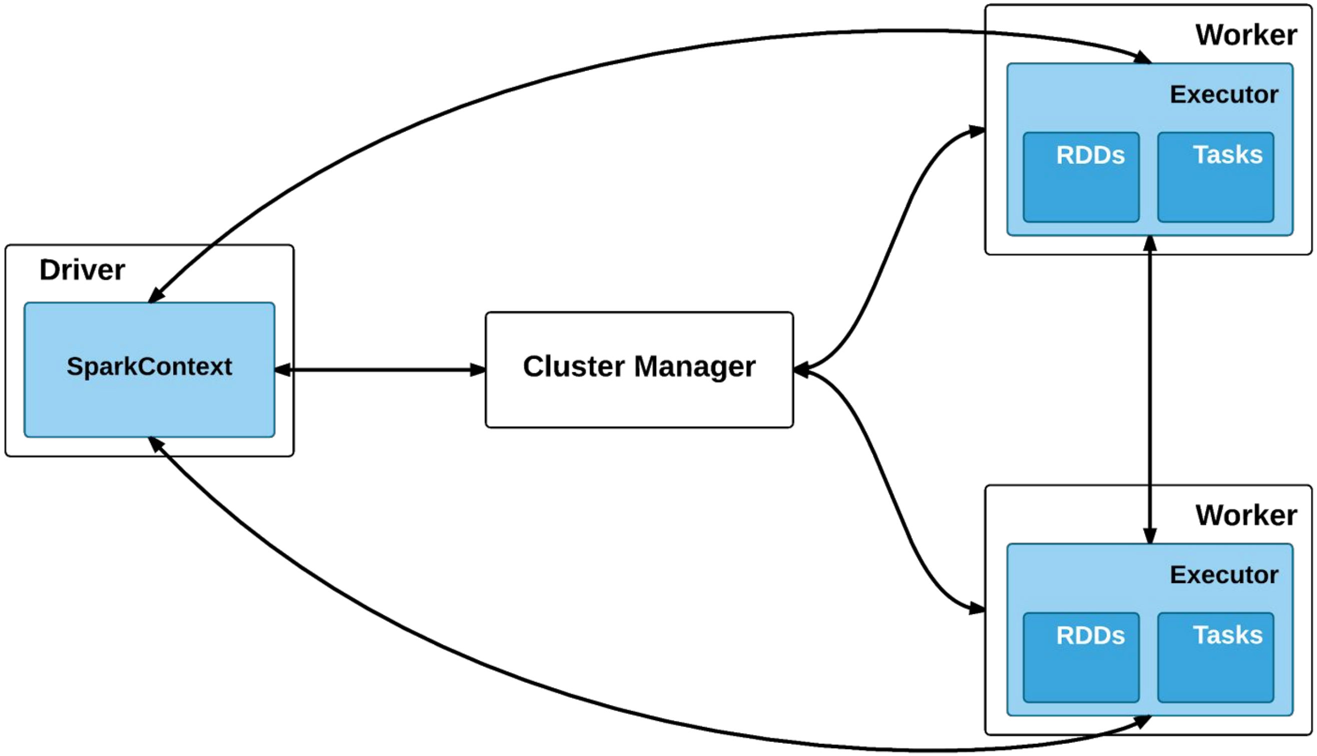
- Local
- Standalone
- Yarn
- Mesos

Architecture - Deploy Modes

Different deploy modes:

- **Local (our laptops)**
- **Standalone (example on next slide)**
- Yarn
- Mesos

Architecture - Cluster Mode



some important 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

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

Official documentation on self-contained applications

Excellent (6 hour) presentation on advanced Spark architecture and features by Sameer Farooqui:
https://www.youtube.com/watch?v=7ooZ4S7Ay6Y