Web: Inceptez.com Mail: info@inceptez.com Call: 7871299810, 7871299817

APACHE SPARK

INTRODUCTION

COURSE HIGHLIGHTS

- Extensive, In-depth & Comprehensive
- Use cases and Hands on based
- Designed as per market requirements that covers
  - Data Engineering
  - Data Analytics
  - Realtime Streaming
  - In-memory processing
  - Columnar & Document Datastore
  - Visualization and Dashboard
- Latest Versions
  - Spark 2.0.1 - Scala 2.11.1 - Java 1.8.0 - Hadoop 2.7.1 - Hive 2.0.1 - Cassandra 3.9.0 - ElasticSearch 5.0.1 - Kibana 5.0.1 - Kafka 0.8 - Centos 7
- End to End System Integration
  - Acquire -> Transport -> Queue -> Transform -> Enrich -> Lookup -> Stream -> Load -> Visualize
- Performance Tuning

COURSE AGENDA

- BIG DATA, Hadoop & Spark Overview
- Spark Basics
- Spark Execution Model
- Working with RDDs
- Spark Essentials
- Spark Distribution Setup and Configurations
- Running Spark on Cluster
- Writing Spark Applications
- Scala and its programming implementation
- RDD Operations in detail
- Interactive Data Analysis with Spark Shell
- Spark API for different File Formats & Compression Codecs
- Caching and Persistence
- Improving Spark Performance
- Spark Core Realtime Use cases
- Exploring Spark SQL
- SQL Realtime Usecases
- Exploring Spark Streaming
- Streaming Realtime Usecases
- Kafka Messaging Queue
- Kafka Realtime Usecases
- Elastic Search Document Datastore
- Elastic Search Realtime Usecases
- Kibana
- Kibana Realtime Usecases
- End to End Project on realtime Fleet tracking analysis with spark streaming application, Kafka, NIFI, Cassandra, Elastic Search and Kibana Dashboard

Use cases & Projects that we execute in this complete course

- Retail Banking
- Server Log Analysis
- Consumer - Product sales analysis
- Federated Data lake
- Twitter popular hashtag trending analysis
- Sensor data streaming pipeline
- Streaming Log files storage
- Desktop Monitoring usecase
- End to End Project on realtime Fleet tracking analysis with spark streaming application, Kafka, NIFI, Cassandra, Elastic Search and Kibana Dashboard.

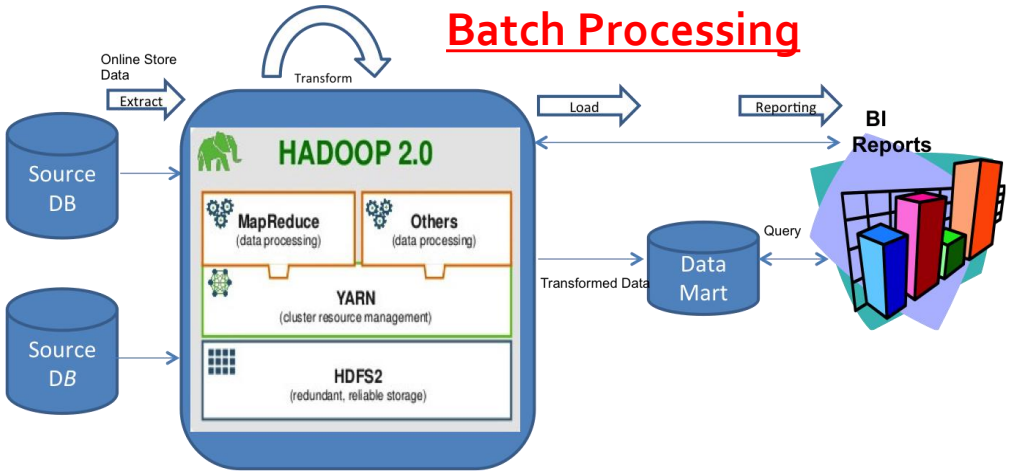
BIG DATA, HADOOP & SPARK OVERVIEW

BIG DATA

- Introduction
- Characteristics

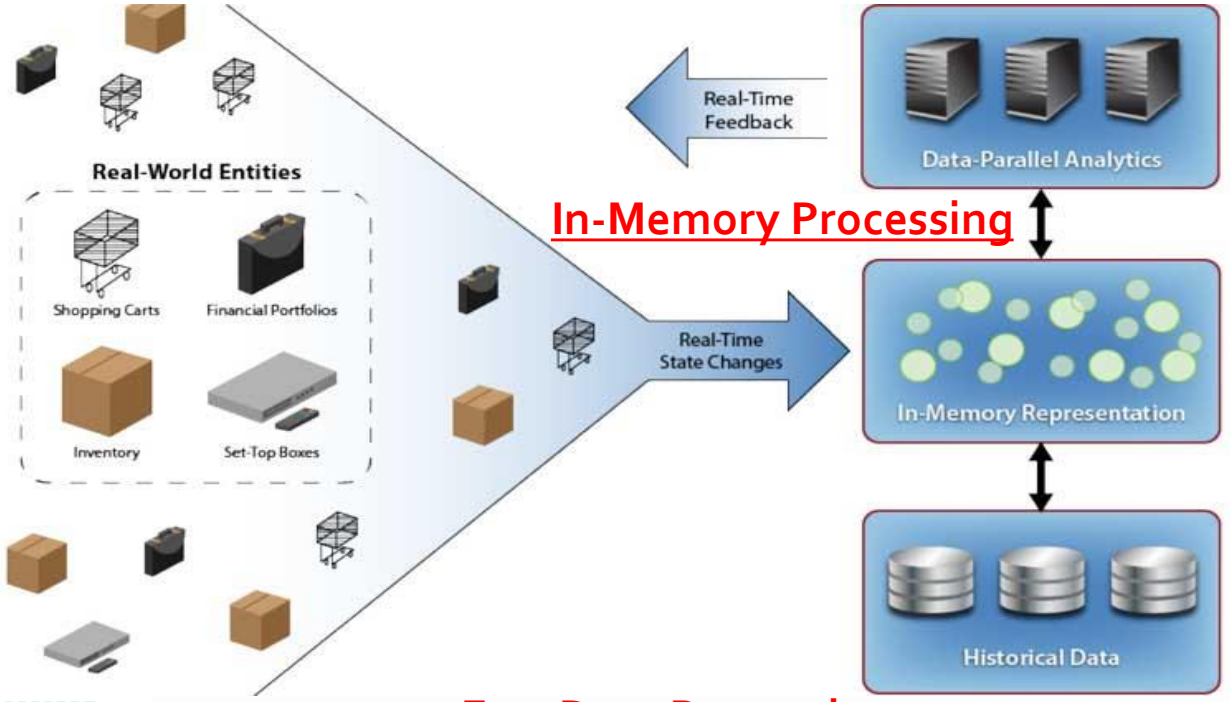
Big Data Engineering Techniques

Batch Processing

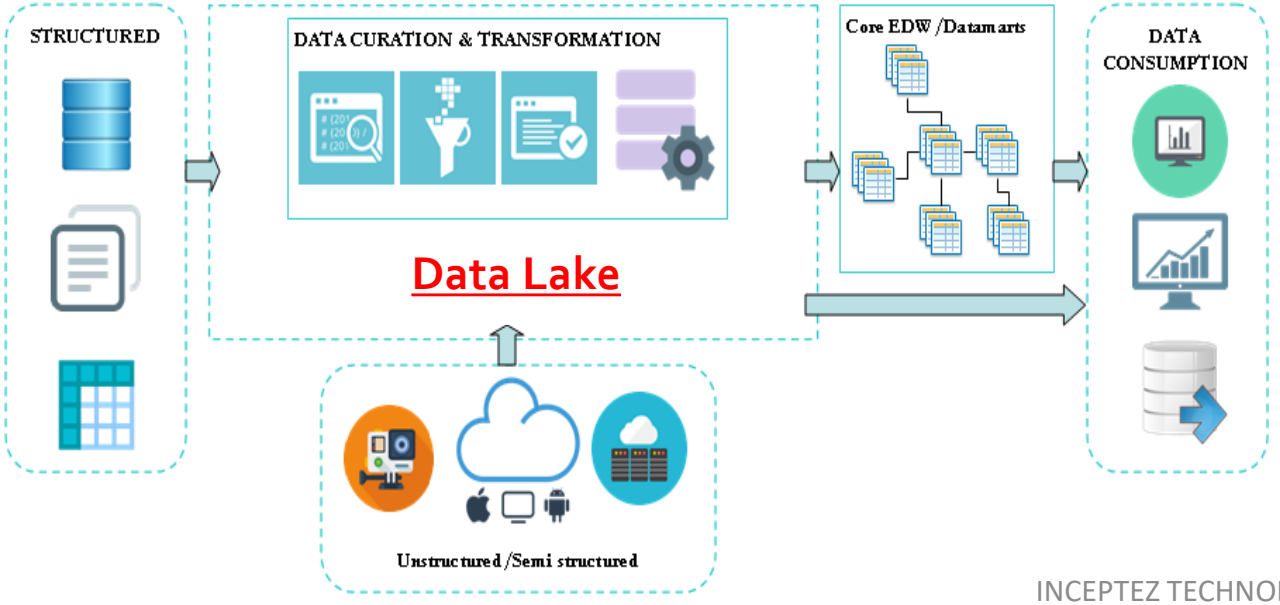


The Big Data ETL

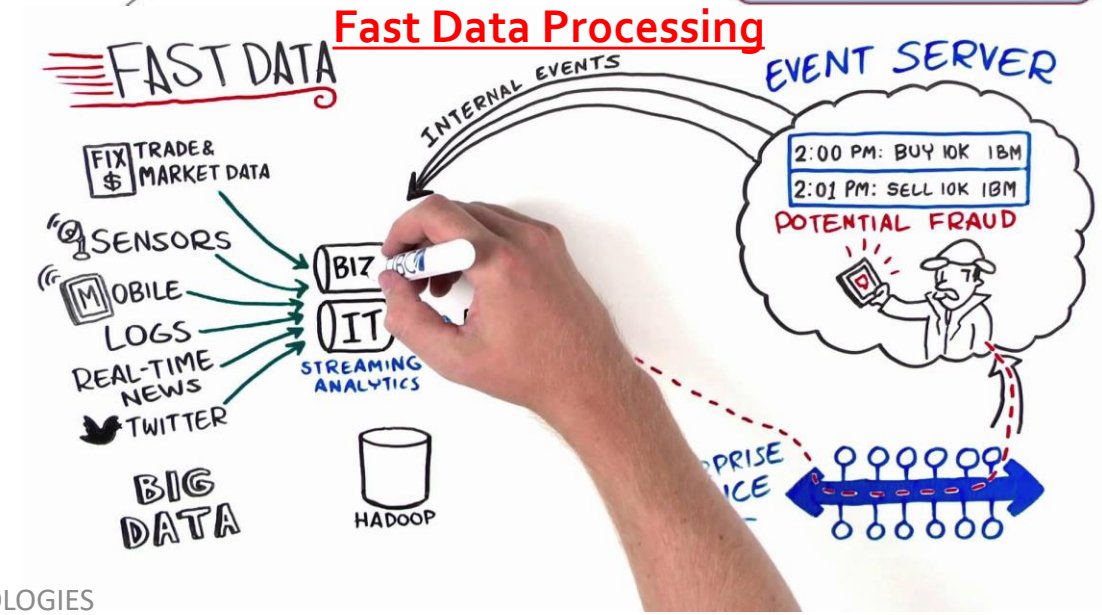
In-Memory Processing



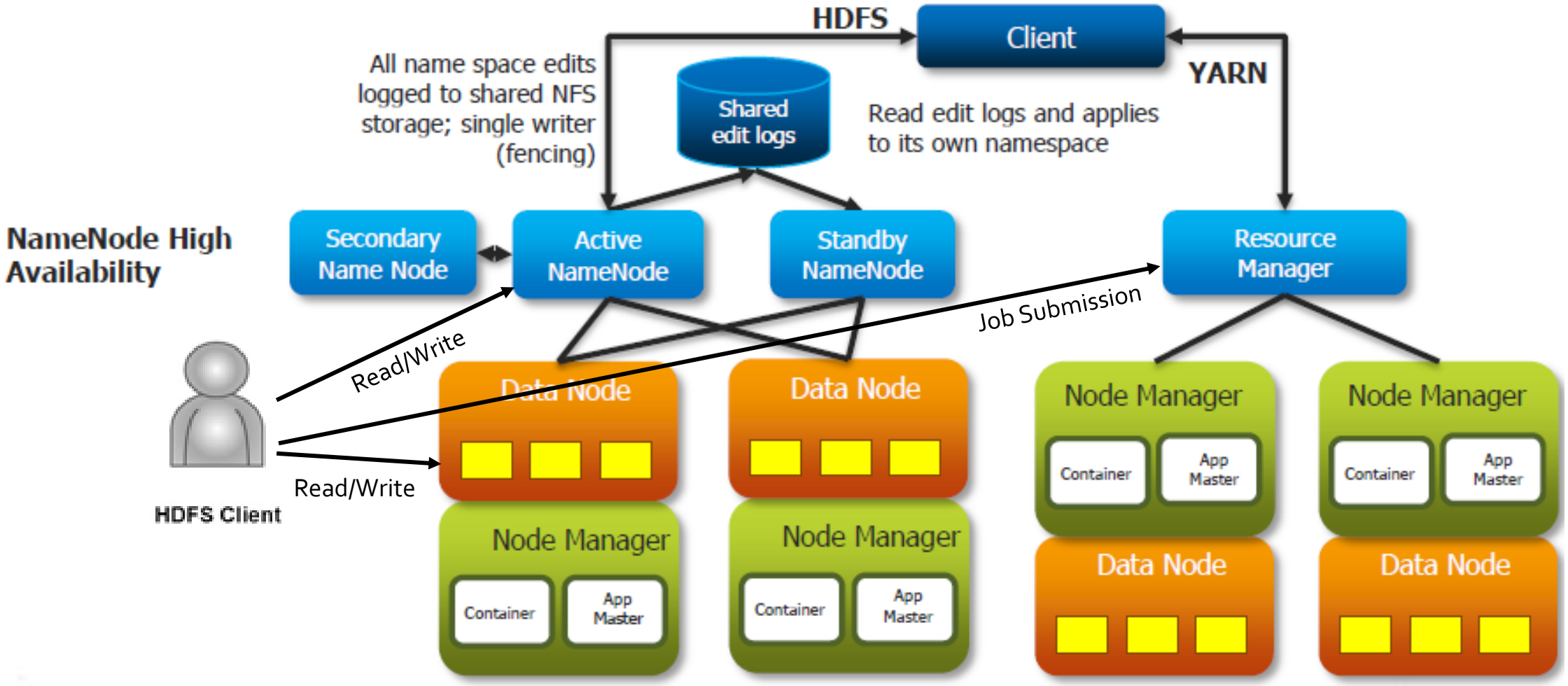
Data Lake



Fast Data Processing

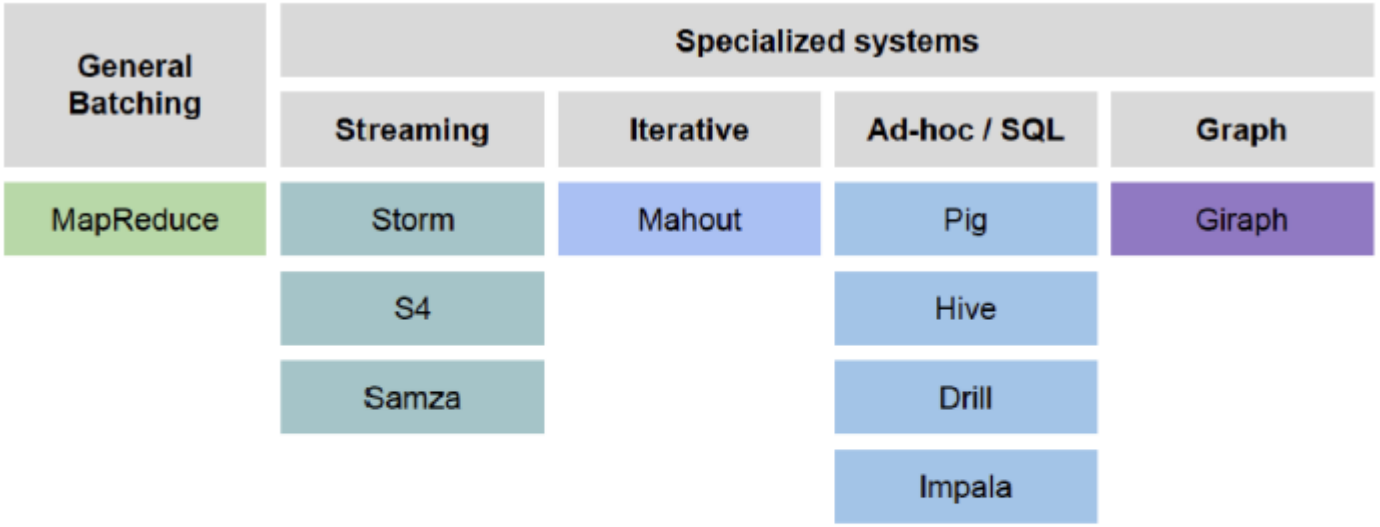


Hadoop – A Quick Overview



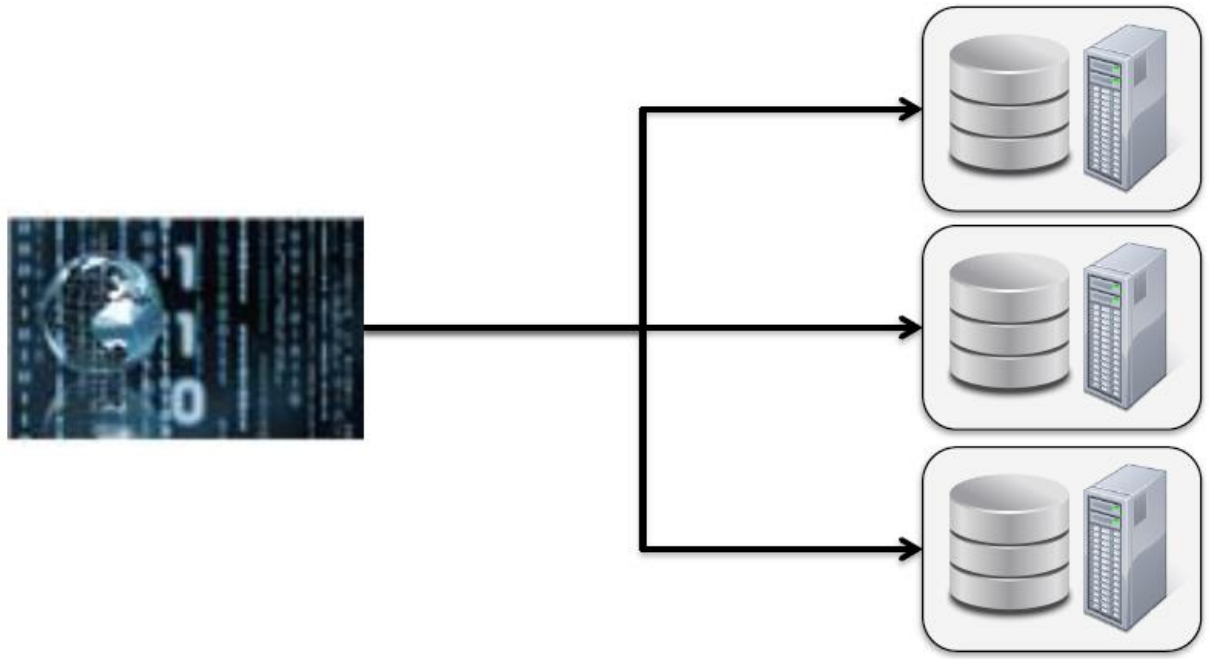
Hadoop Ecosystems – No Unified Vision

- Sparse Modules
- More of Batch
- Diversity of Tools/APIs
- Huge coding efforts
- Heavily I/O Bounded
- High Latency



Hadoop & Spark Approach

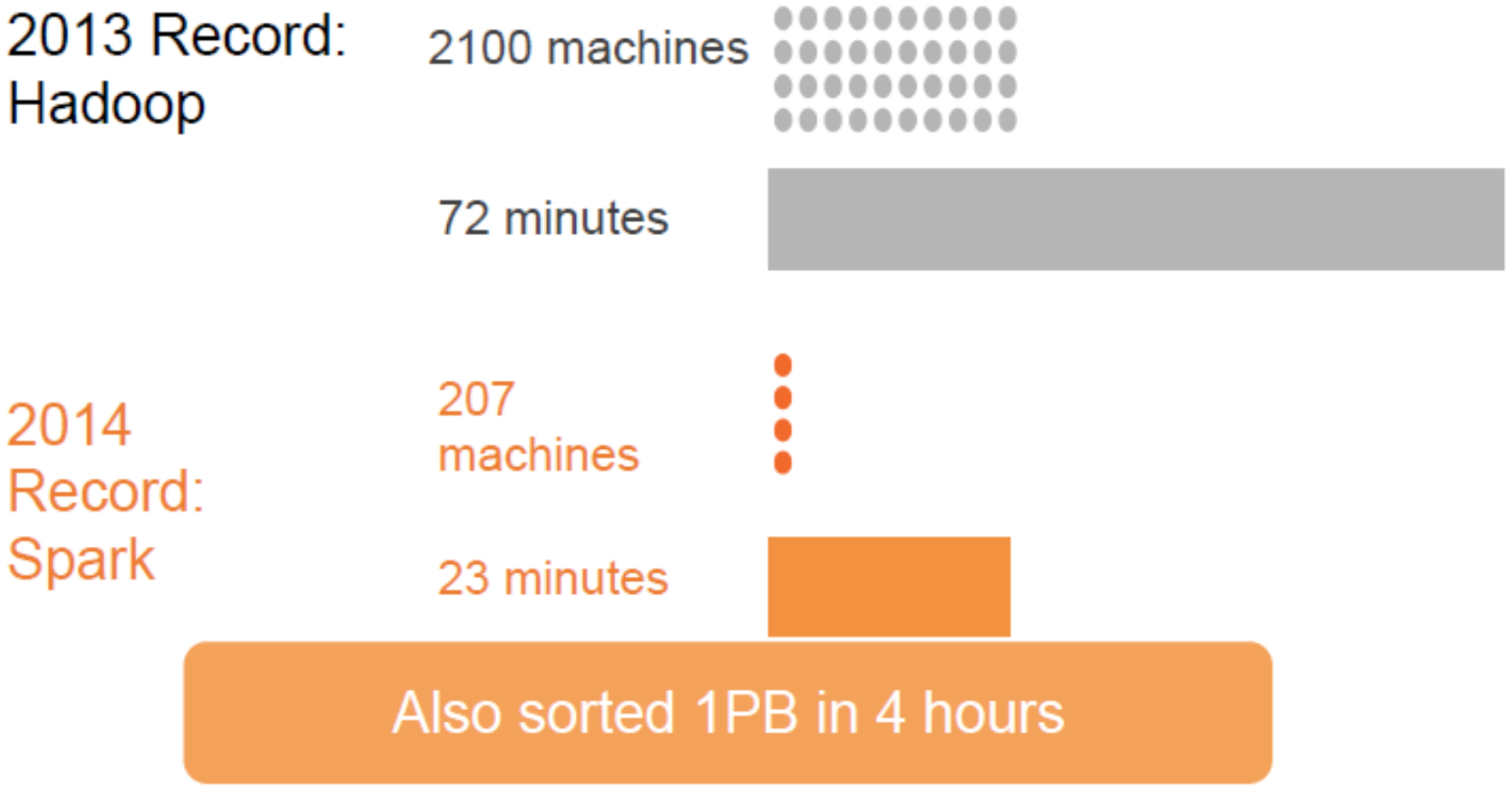
- **Hadoop introduced a radical new approach based on two key concepts**
  - Distribute the data when it is stored
  - Run computation where the data is
- **Spark takes this new approach to the next level**
  - Data is distributed in memory



On-Disk Sort Record:

Time to sort 100TB

"Run programs up to 100x faster than Hadoop MapReduce in memory, or 10x faster on disk".

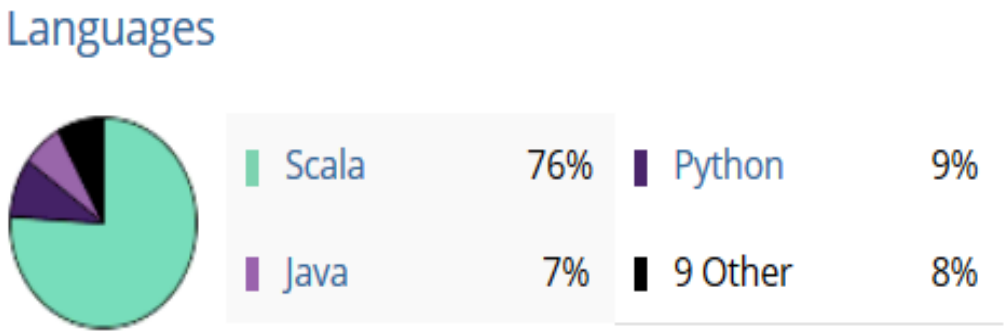


SPARK BASICS

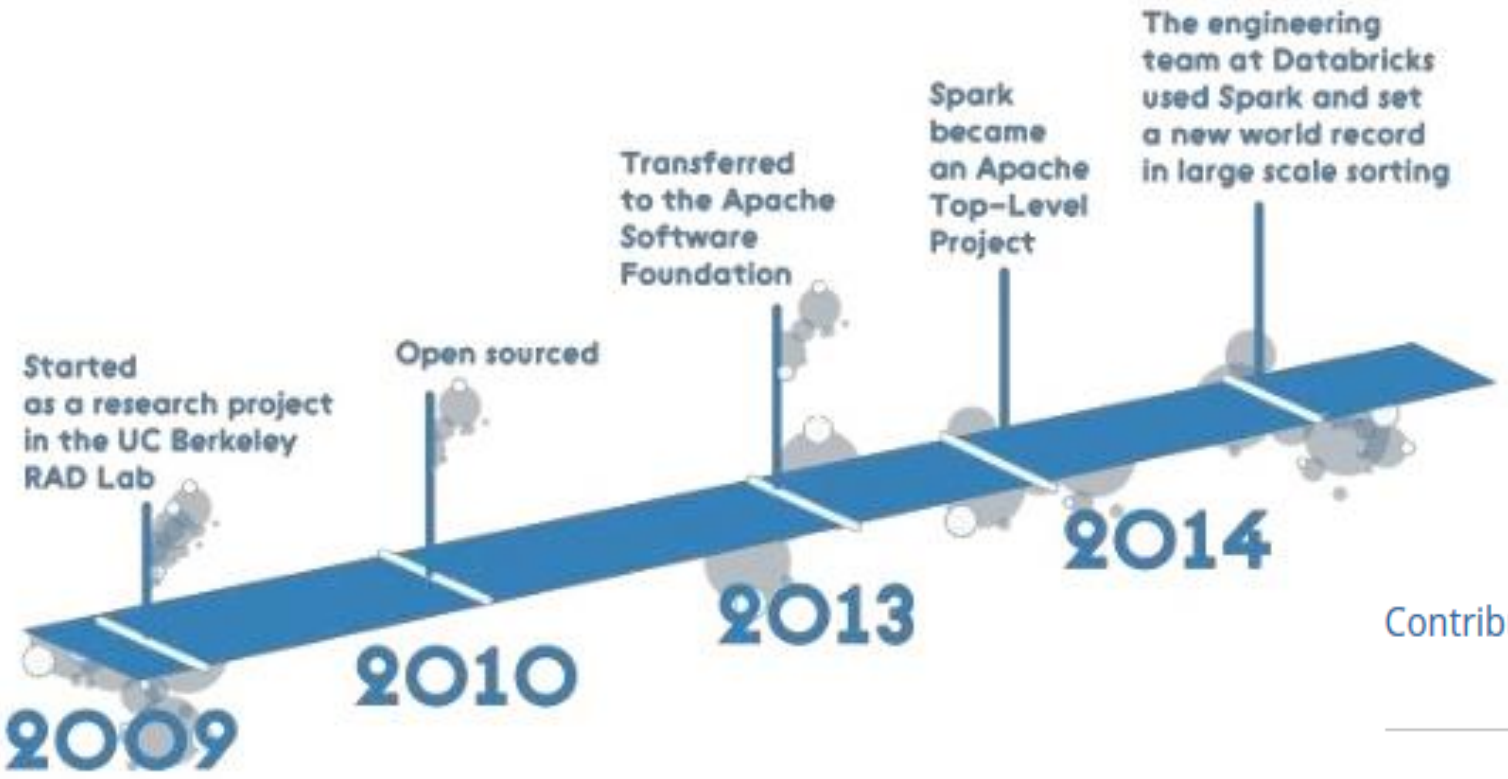
UNIFIED BIG DATA PROCESSING ENGINE

Brief Introduction to Apache Spark

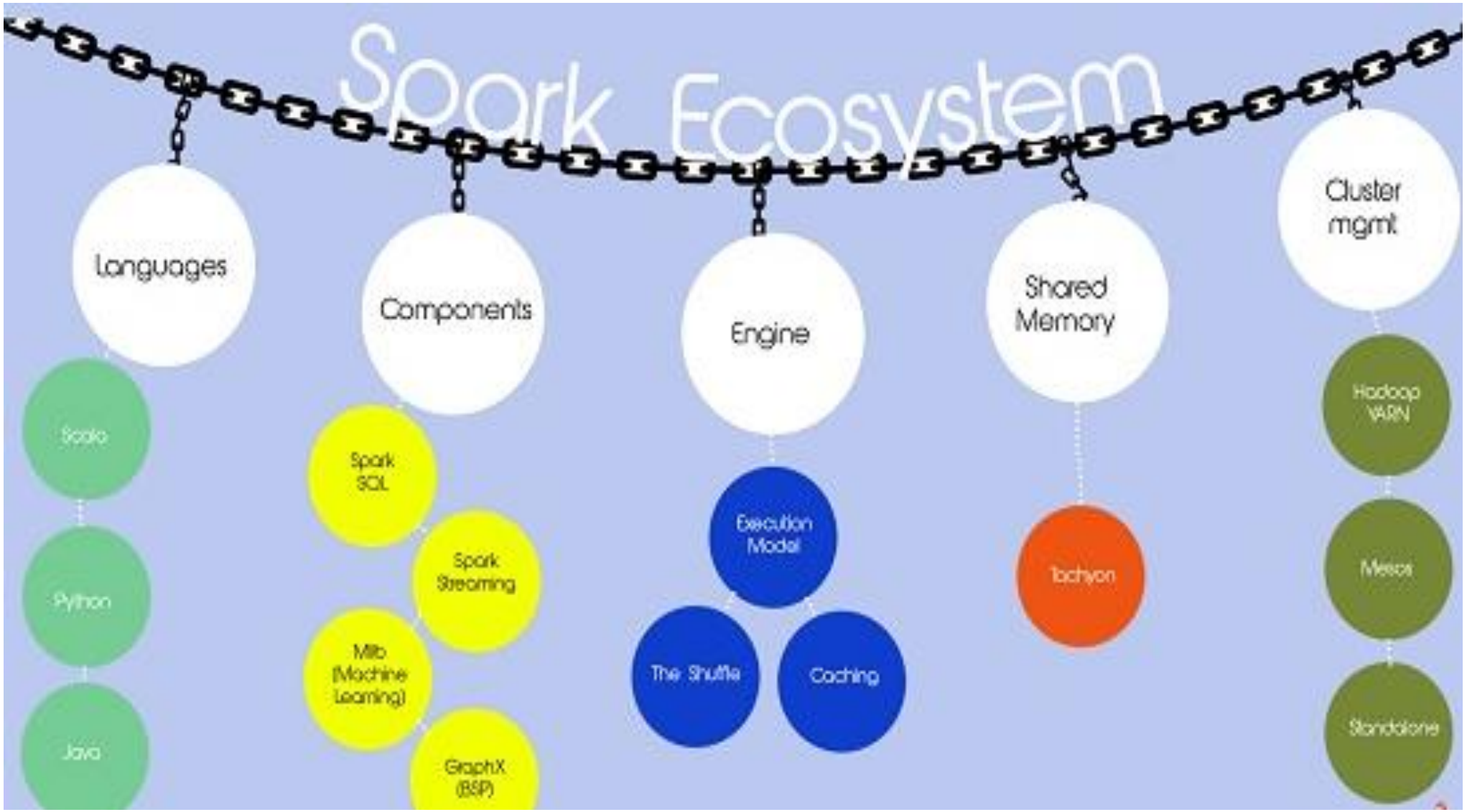
- Apache Spark™ is a fast and general engine for large-scale data processing framework for massive parallel computing (cluster)
- Computational Engine - scheduling, distributing, managing, and monitoring application
- Harnessing power of cheap memory
- Written using Scala, Java and Python languages
- High-level APIs support in Scala, Java and Python
- Developers from 50+ companies includes UC Berkley, Cloudera, Yahoo, Databricks, Intel, Groupon etc.,



Brief History



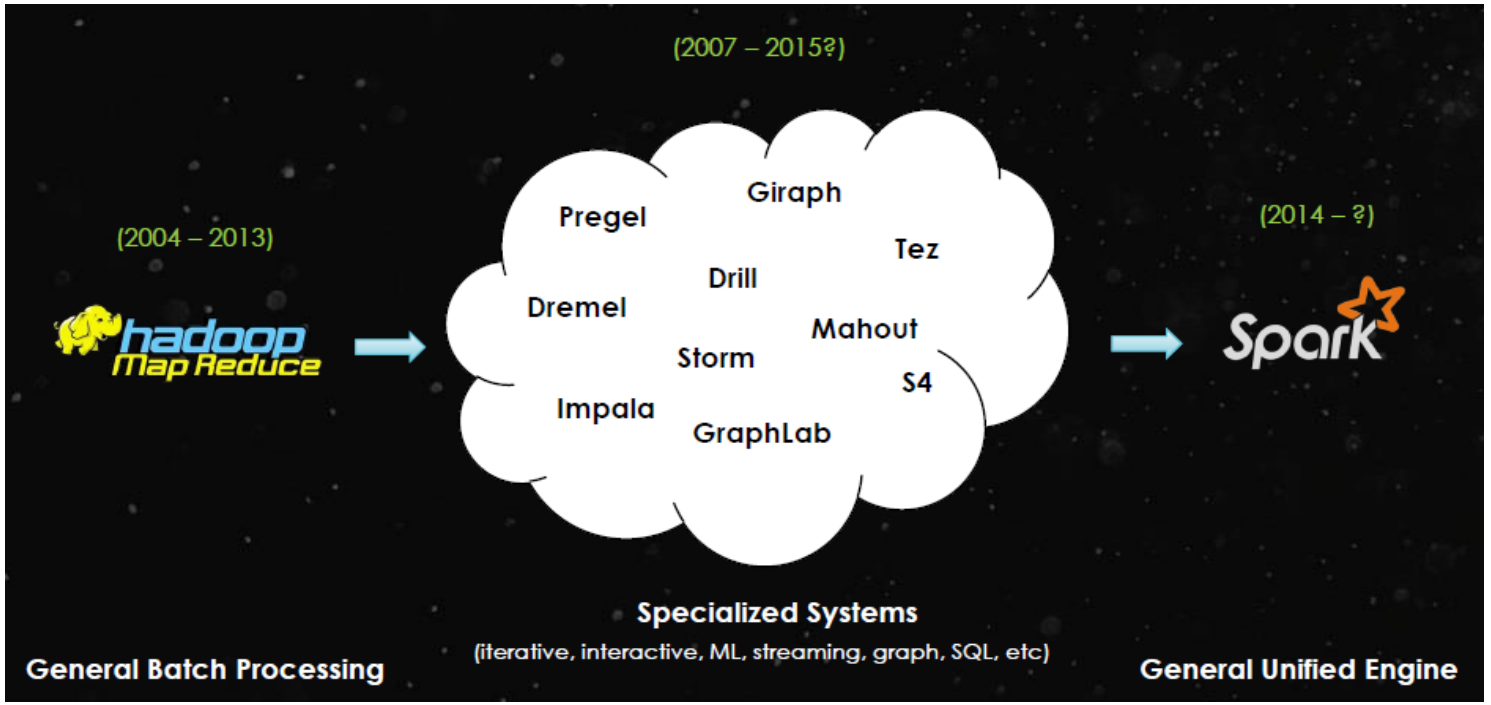
Ecosystems


Spark Ecosystem
Languages
Components
Engine
Shared Memory
Cluster mgmt
Scala
Python
Java
Spark SQL
Spark Streaming
Mlib (Machine Learning)
GraphX (BSP)
Execution Model
The Shuffle
Caching
Tachyon
Hadoop YARN
Mesos
Standalone

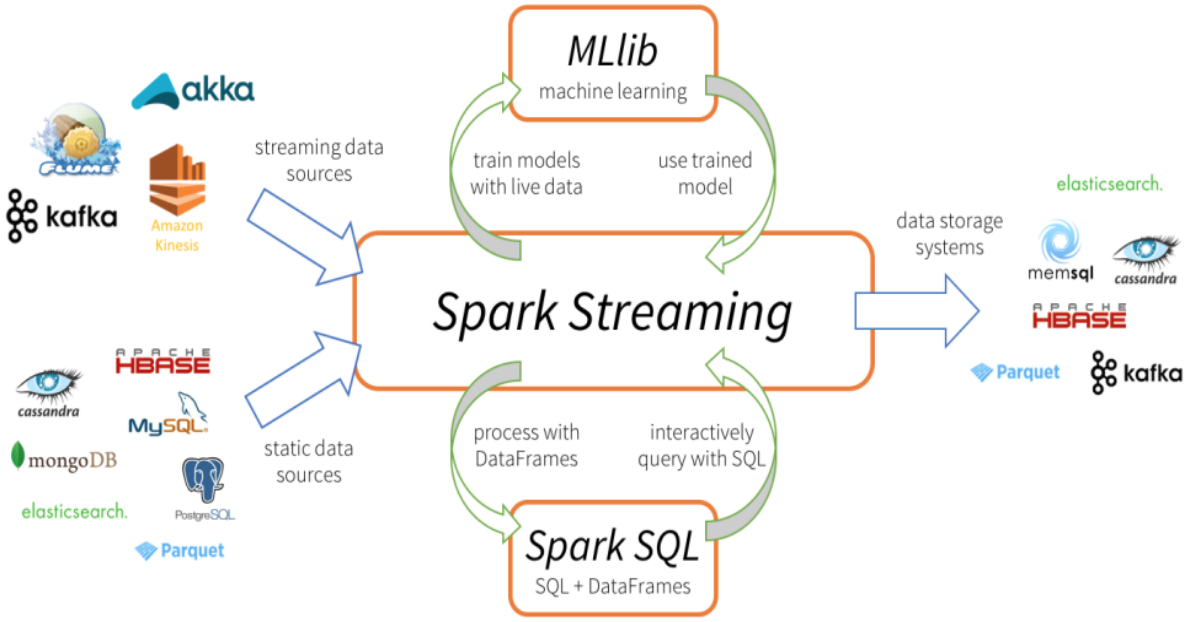
Why Spark

- Distributed data analytics Unified system
- Scalability & Fault Tolerant - Proven to be scalable over 8000 nodes in production.
- Interactive Shell - Interactive command line interface (in Scala or Python)
- No copying or ETL of data between systems
- Combine processing types in one program
- Code reuse
- One system to learn
- One system to maintain
- Realtime, Iterative, In memory

Unified Engine

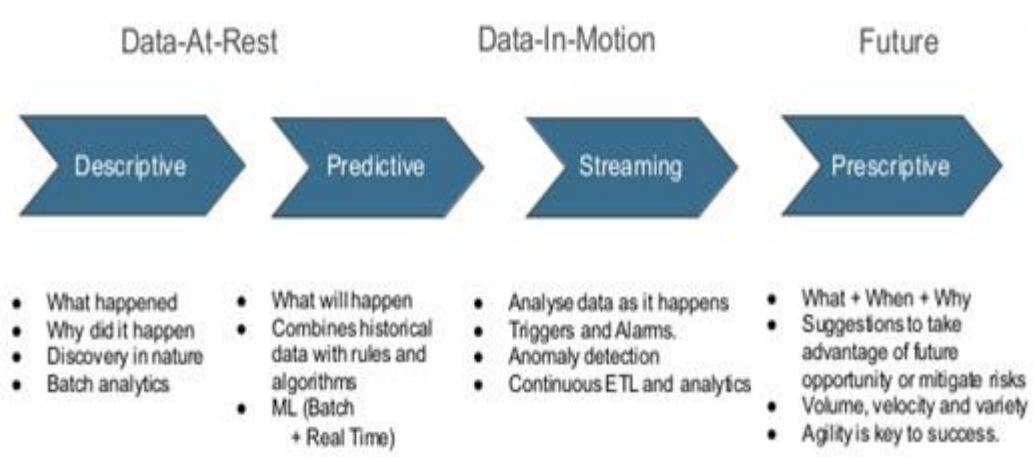


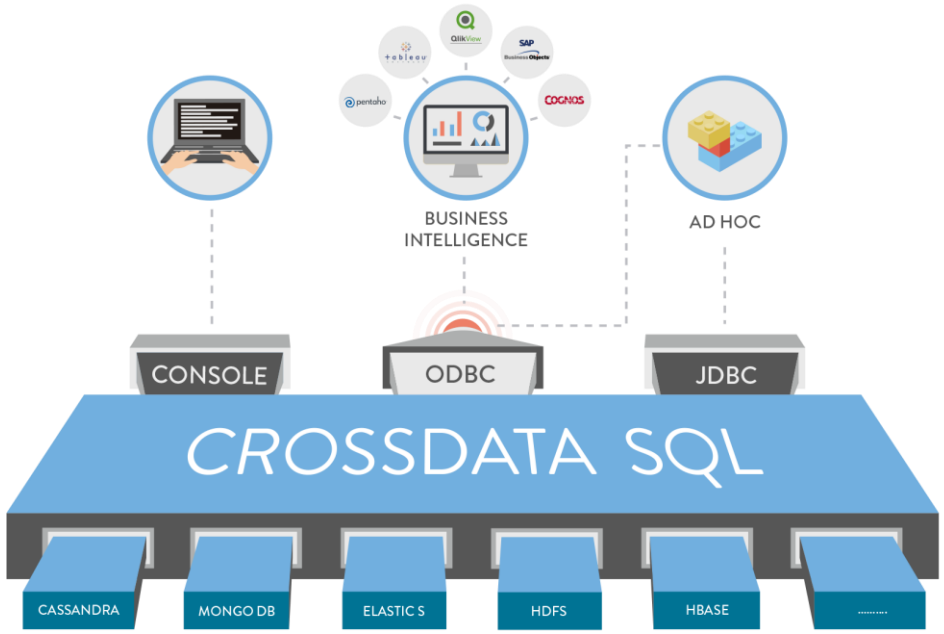
Technical Benefits



- Alternative to MapReduce
- Compatibility with HDFS, HBASE and YARN
- Spark SQL component to access structured data
- Stable API
- Support for multiple languages
- Library support

Business Benefits



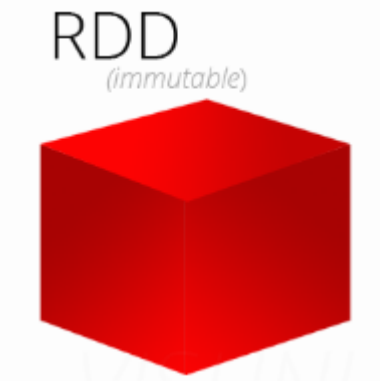


SPARK EXECUTION MODEL

Pillars of Spark

RDD (Resilient Distributed Dataset)

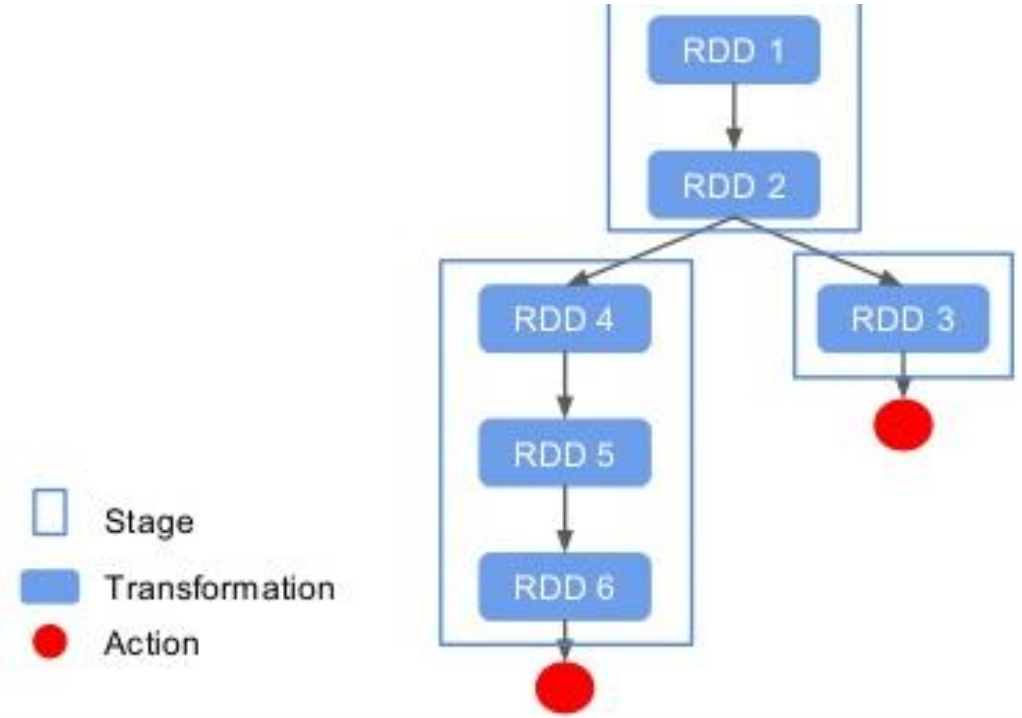
- **RDD (Resilient Distributed Dataset)**
  - Resilient – if data in memory is lost, it can be recreated
  - Distributed – stored in memory across the cluster
  - Dataset – initial data can come from a file or be created programmatically
- **RDDs are the fundamental unit of data in Spark**
- **Most Spark programming consists of performing operations on RDDs**



Pillars of Spark

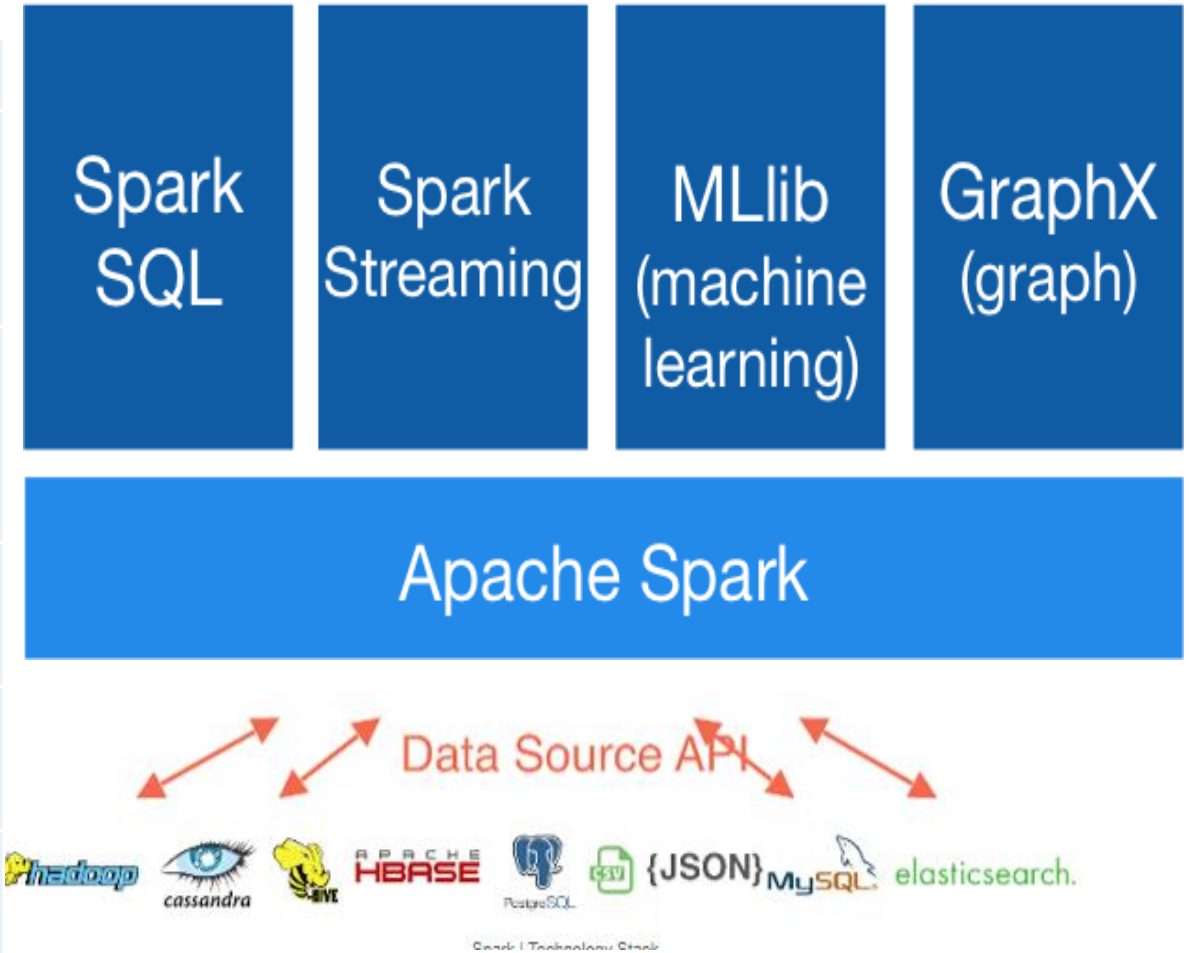
Direct Acyclic Graph – sequence of computations performed on data

- *Node* – RDD partition
- *Edge* – transformation on top of data
- *Acyclic* – graph cannot return to the older partition
- *Direct* – transformation is an action that transitions data partition state (from A to B)



Spark Library Stack

Library	Use	Supported Languages
Spark Core	Base Engine, scheduling, memory management, fault recovery, interacting with storage systems	Scala, Java, Python, and R
Spark SQL	Enables the use of SQL statements or DataFrame API inside Spark applications	Scala, Java, Python, and R
Spark Streaming	Enables processing of live data streams	Scala, Java, and Python
Spark MLlib	Enables development of machine learning applications	Scala, Java, Python, and R
Spark GraphX	Enables graph processing and supports a growing library of graph algorithms	Scala

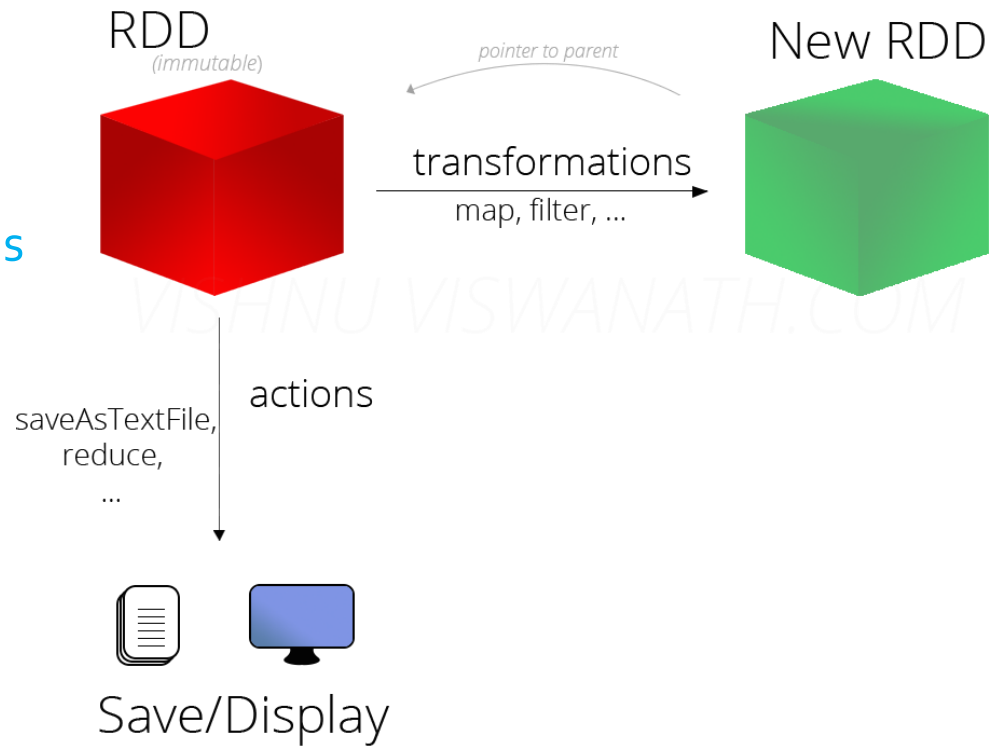


WORKING WITH RDDS

All about RDDs

- Core Spark abstraction
- Lazily evaluated
- Partitions can be persisted in-memory or on-disk
- Broken up into partitions, which are distributed across nodes
- Abstraction to represent the large distributes
- Immutable, re-computable
- Fault tolerant – Via concept of Lineage
- Contains transformation history ("lineage") for data set

- Partitions – Set of data splits associated with this RDD
- Dependencies – List of parent RDDs involved in computation
- Compute – Function to compute partitions
- Preferred Locations – list of location for input split to compute



Creating RDDs

Three ways to create an RDD

- **From a file or set of files**
- **From data in memory**
- **From another RDD**

Types of RDDs

- **HadoopRDD**
- **FilteredRDD**
- **MappedRDD**
- **PairRDD**
- **ShuffledRDD**
- **UnionRDD**
- **JsonRDD**
- **SchemaRDD**
- **CassandraRDD**
- **EsSparkRDD**

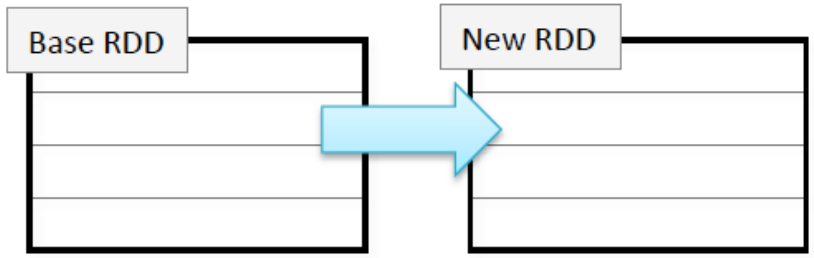
Spark Operations =  TRANSFORMATIONS

\+

 ACTIONS

RDD Operations - Transformations

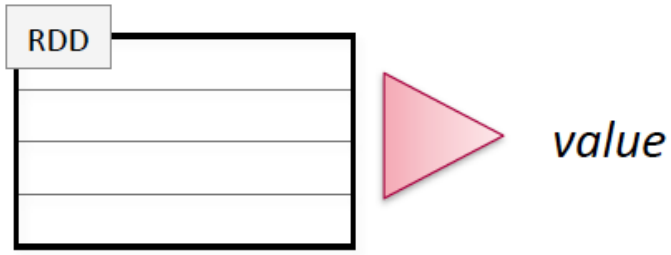
- **Transformations create a new RDD from an existing one**
- **RDDs are immutable**
  - Data in an RDD is never changed
  - Transform in sequence to modify the data as needed



- **Some common transformations**
  - **`map(`***`function`***`)`** – creates a new RDD by performing a function on each record in the base RDD
  - **`filter(`***`function`***`)`** – creates a new RDD by including or excluding each record in the base RDD according to a boolean function

RDD Operations - Actions

- **Some common actions**
  - **`count()`** – return the number of elements
  - **`take(`***`n`***`)`** – return an array of the first *n* elements
  - **`collect()`** – return an array of all elements
  - **`saveAsTextFile(`***`filename`***`)`** – save to text file(s)



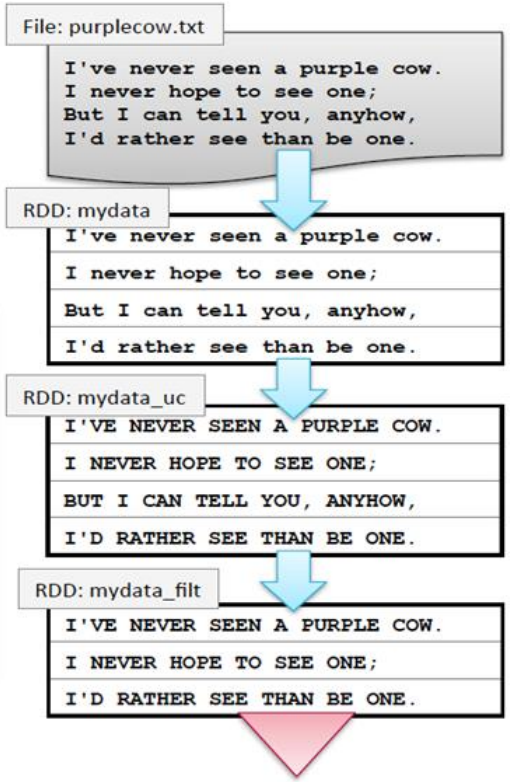

```
> mydata =
  sc.textFile("purplecow.txt")

> mydata.count()
4

> for line in mydata.take(2):
    print line
I've never seen a purple cow.
I never hope to see one;
```

```
> val mydata =
  sc.textFile("purplecow.txt")

> mydata.count()
4

> for (line <- mydata.take(2))
    println(line)
I've never seen a purple cow.
I never hope to see one;
```

RDD Execution model – Lazy Evaluation

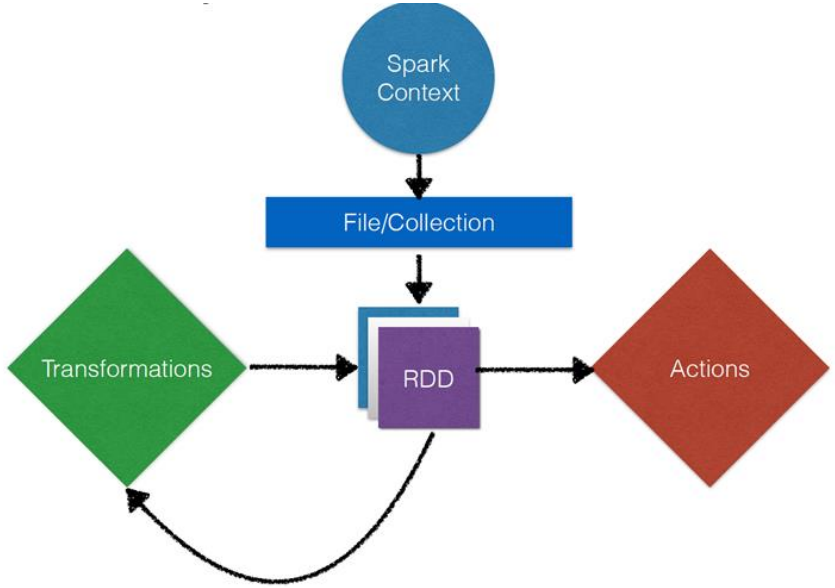
- **RDDs are not always immediately materialized**
  - Spark logs the *lineage* of transformations used to build datasets
- **Data in RDDs is not processed until an *action* is performed**
  - RDD is materialized in memory upon the first action that uses it

```
> mydata = sc.textFile("purplecow.txt")
> mydata_uc = mydata.map(lambda line:
  line.upper())
> mydata_filt = \
     mydata_uc.filter(lambda line: \
     line.startswith('I'))
> mydata_filt.count()
3
```

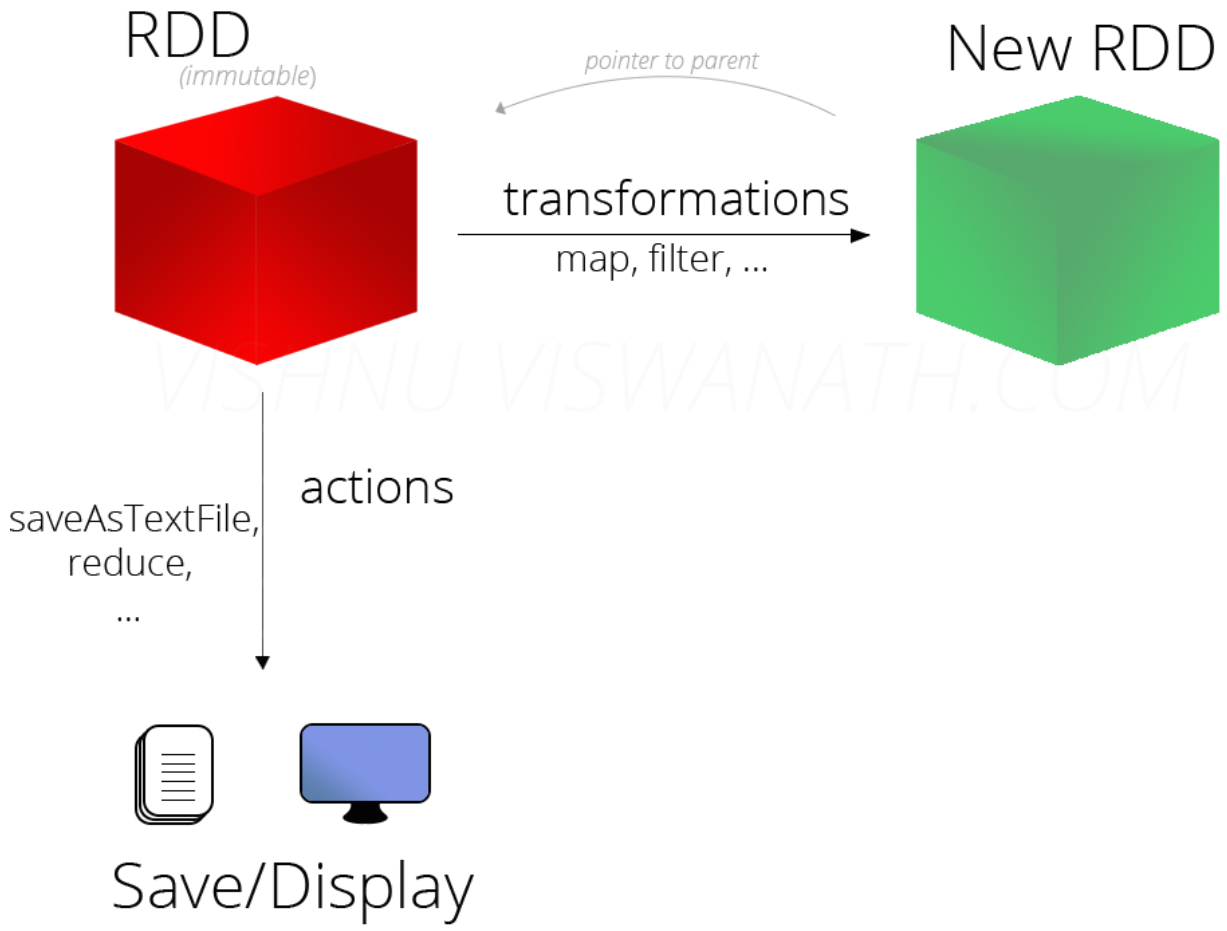


Lazy Evaluation

Spark Context - Main entry point for Spark functionality, Represents a connection to a Spark cluster

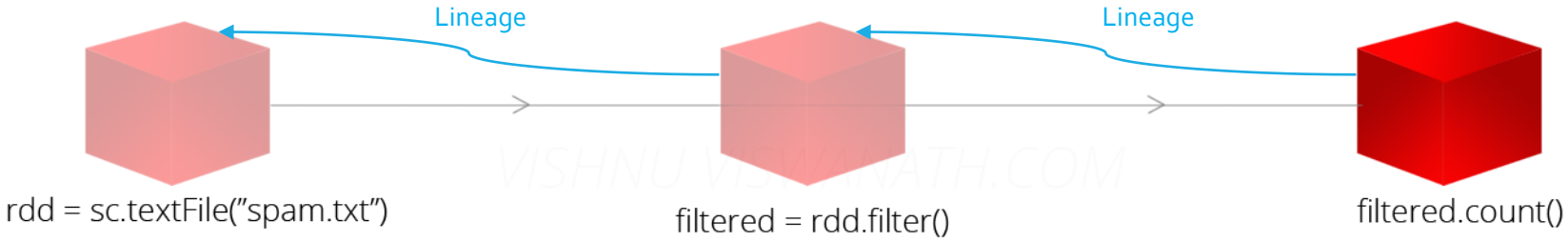


RDD Execution Model



RDD Lineage

- When you call a transformation, Spark does not execute it immediately, instead it creates a **lineage**.
- A lineage keeps track of what all transformations has to be applied on that RDD, including from where it has to read the data.

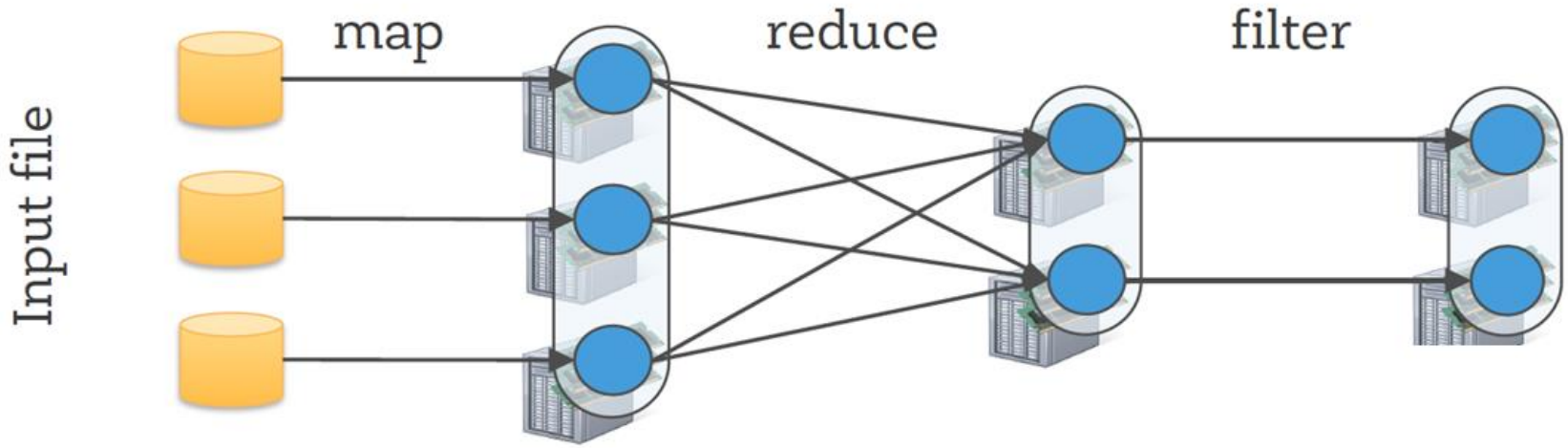



```
val rdd = sc.textFile("spam.txt")
val filtered = rdd.filter(line => line.contains("money"))
filtered.count()
```

Fault Tolerance

RDDs track *lineage* info to rebuild lost data

```
file.map(lambda rec: (rec.type, 1))
    .reduceByKey(lambda x, y: x + y)
    .filter(lambda (type, count): count > 10)
```



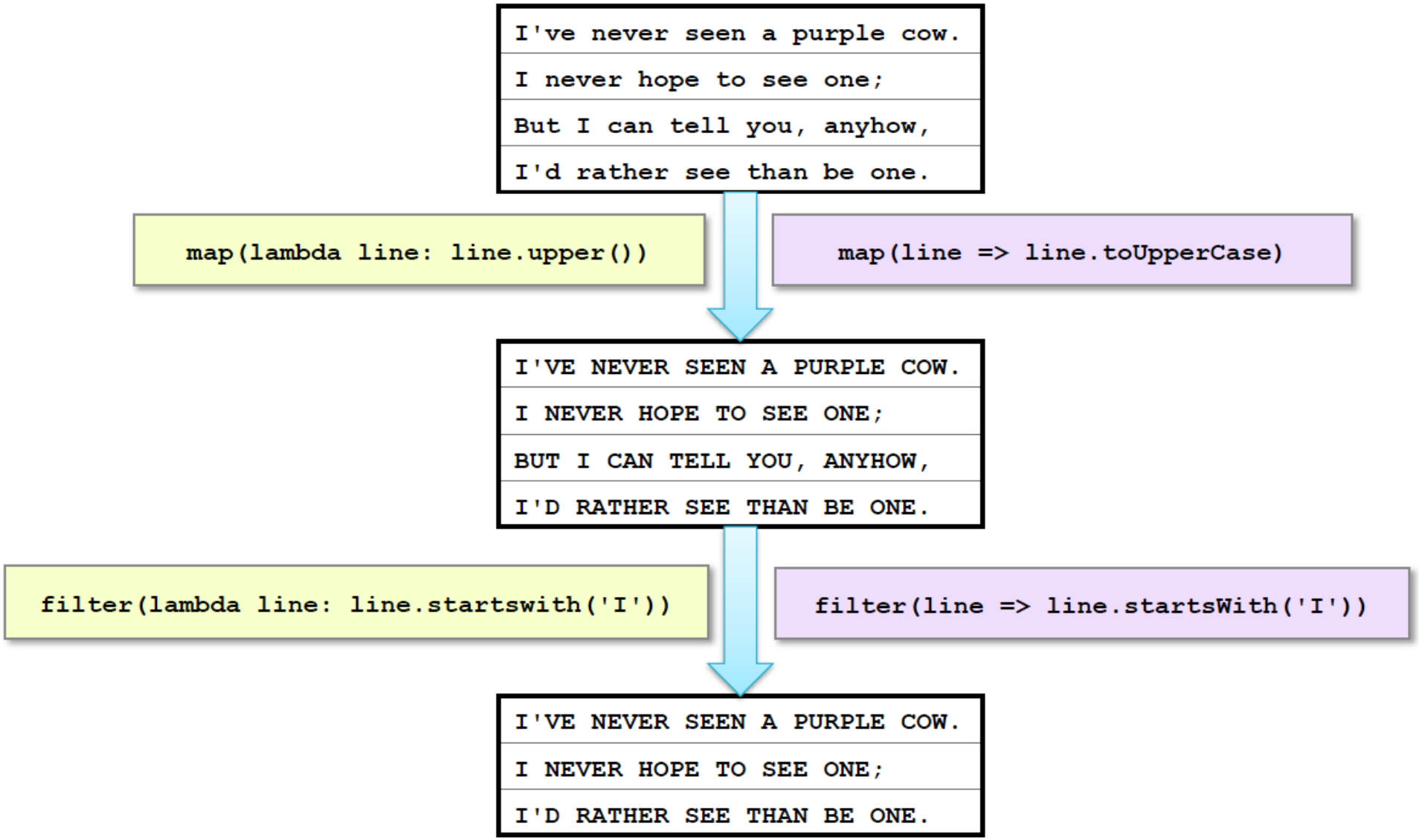
RDD Operations

- **Transformations**
  - apply user function to every element in a partition (or to the whole partition)
  - apply aggregation function to the whole dataset (groupBy, sortBy)
  - introduce dependencies between RDDs to form DAG
  - provide functionality for repartitioning (repartition, partitionBy)
- **Actions**
  - trigger job execution
  - used to materialize computation results
- **Extra: persistence**
  - explicitly store RDDs in memory, on disk or off-heap (cache, persist)
  - checkpointing for truncating RDD lineage

Loading data in RDD

- **For file-based RDDS, use `SparkContext.textFile`**
  - Accepts a single file, a wildcard list of files, or a comma-separated list of files
  - Examples
    - `sc.textFile("myfile.txt")`
    - `sc.textFile("mydata/*.log")`
    - `sc.textFile("myfile1.txt,myfile2.txt")`
  - Each line in the file(s) is a separate record in the RDD
- **Files are referenced by absolute or relative URI**
  - Absolute URI: `file:/home/training/myfile.txt`
  - Relative URI (uses default file system): `myfile.txt`

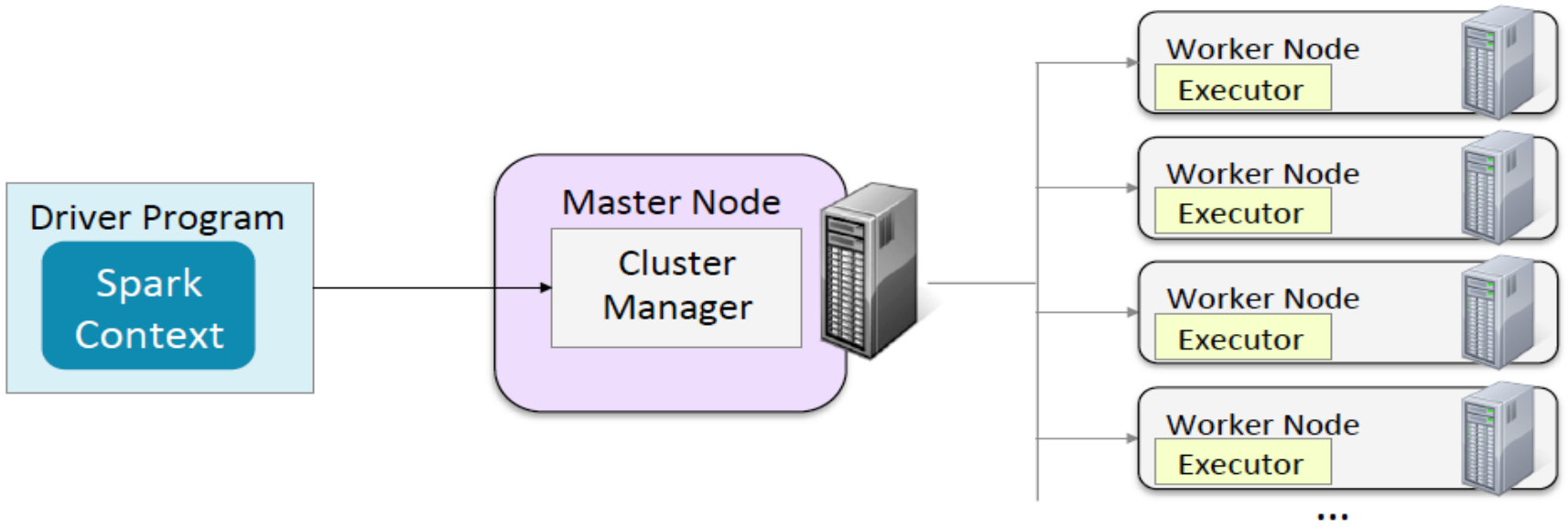
Example: `map` and `filter` Transformations



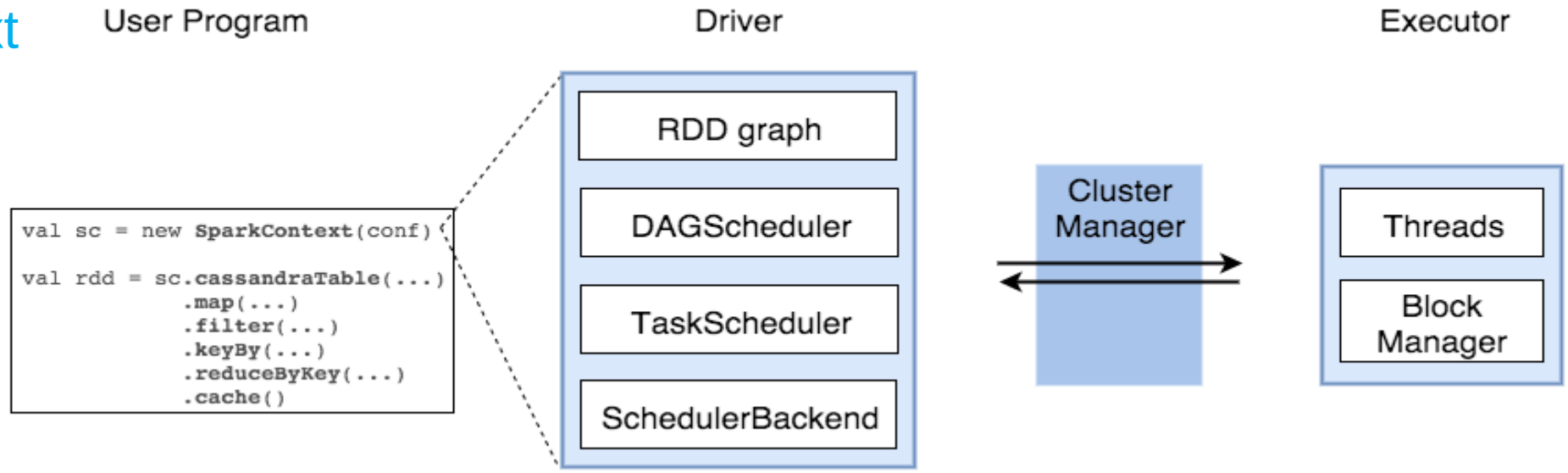
SPARK ESSENTIALS

Spark Terminologies

- **A Spark Driver**
  - The “main” program
    - Either the Spark Shell or a Spark application
  - Creates a Spark Context configured for the cluster
  - Communicates with Cluster Manager to distribute tasks to executors



SparkContext



SparkContext

represents the connection to a Spark cluster, and can be used to create RDDs, accumulators and broadcast variables on that cluster

DAGScheduler

computes a DAG of stages for each job and submits them to TaskScheduler determines preferred locations for tasks (based on cache status or shuffle files locations) and finds minimum schedule to run the jobs

TaskScheduler

responsible for sending tasks to the cluster, running them, retrying if there are failures, and mitigating stragglers

SchedulerBackend

backend interface for scheduling systems that allows plugging in different implementations(Mesos, YARN, Standalone, local)

BlockManager

provides interfaces for putting and retrieving blocks both locally and remotely into various stores (memory, disk, and off-heap)

Spark Configuration

spark-defaults.conf

Application Level Settings

Spark-env.sh

Per Machine Settings

Log4j.properties

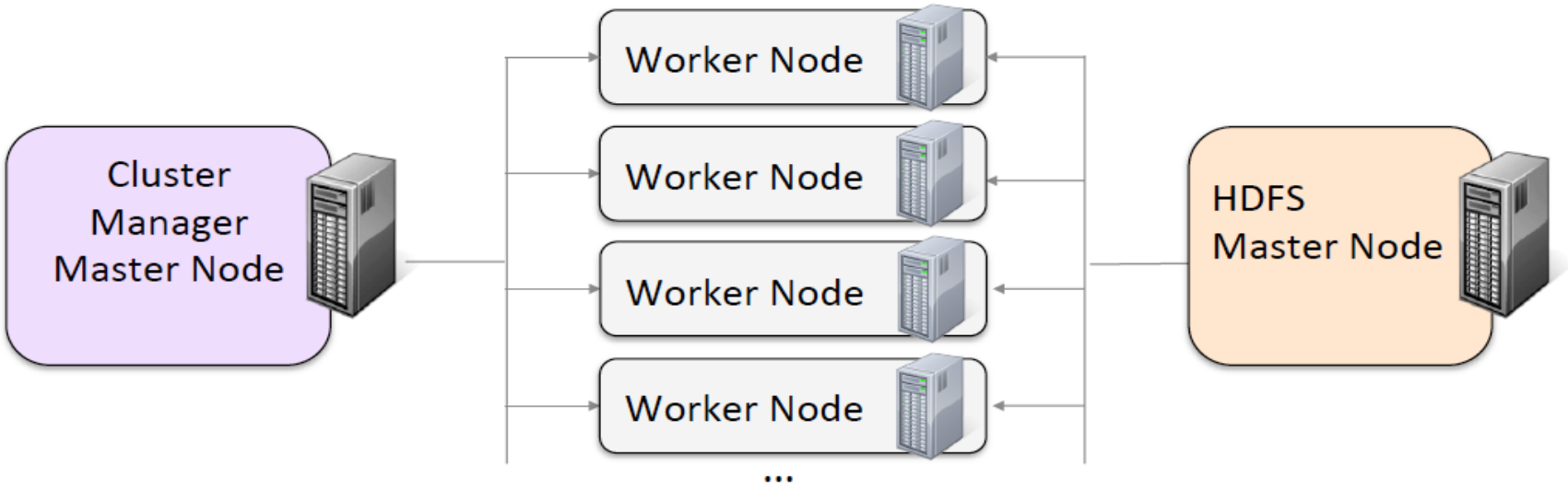
log4j.properties

Spark web UIs

- Spark application UI (contains Stages, Storage, Environment & Executors)
  http://localhost:4040
- Spark Master UI
  Spark Standalone - **http://localhost:7077**
  Yarn - **http://localhost:8088**

Cluster Terminologies

- **A cluster is a group of computers working together**
  - Usually runs HDFS in addition to Spark Standalone, YARN, or Mesos
- **A node is an individual computer in the cluster**
  - *Master* nodes manage distribution of work and data to *worker* nodes
- **A daemon is a program running on a node**
  - Each performs different functions in the cluster

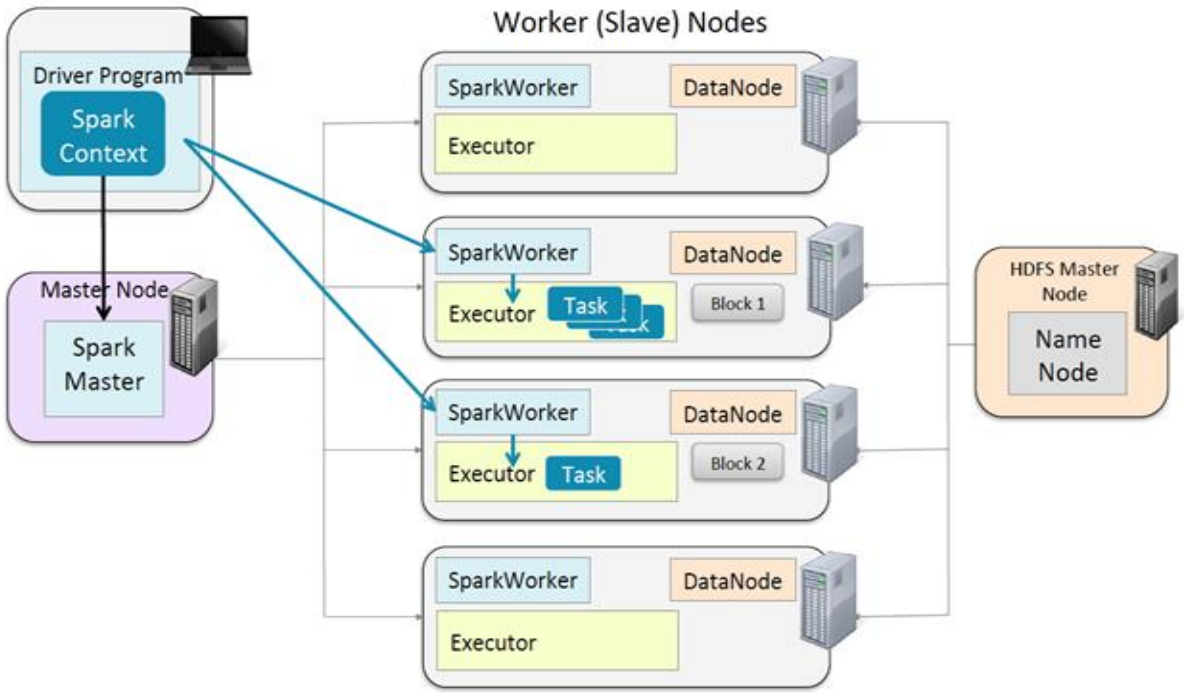


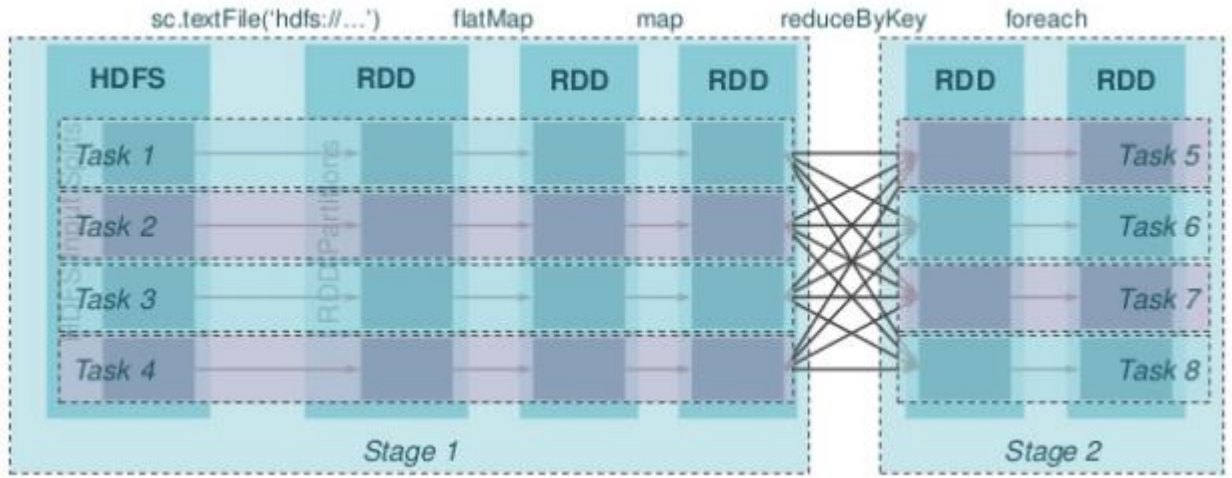
Programming Terminologies

- **Application Jar**
  - User Program and its dependencies except Hadoop & Spark Jars bundled into a Jar file
- **Driver Program**
  - The process to start the execution (main() function)
- **Cluster Manager**
  - An external service to manage resources on the cluster (standalone manager, YARN, Apache Mesos)
- **Deploy Mode**
  - **cluster** : Driver inside the cluster
  - **client** : Driver outside of Cluster

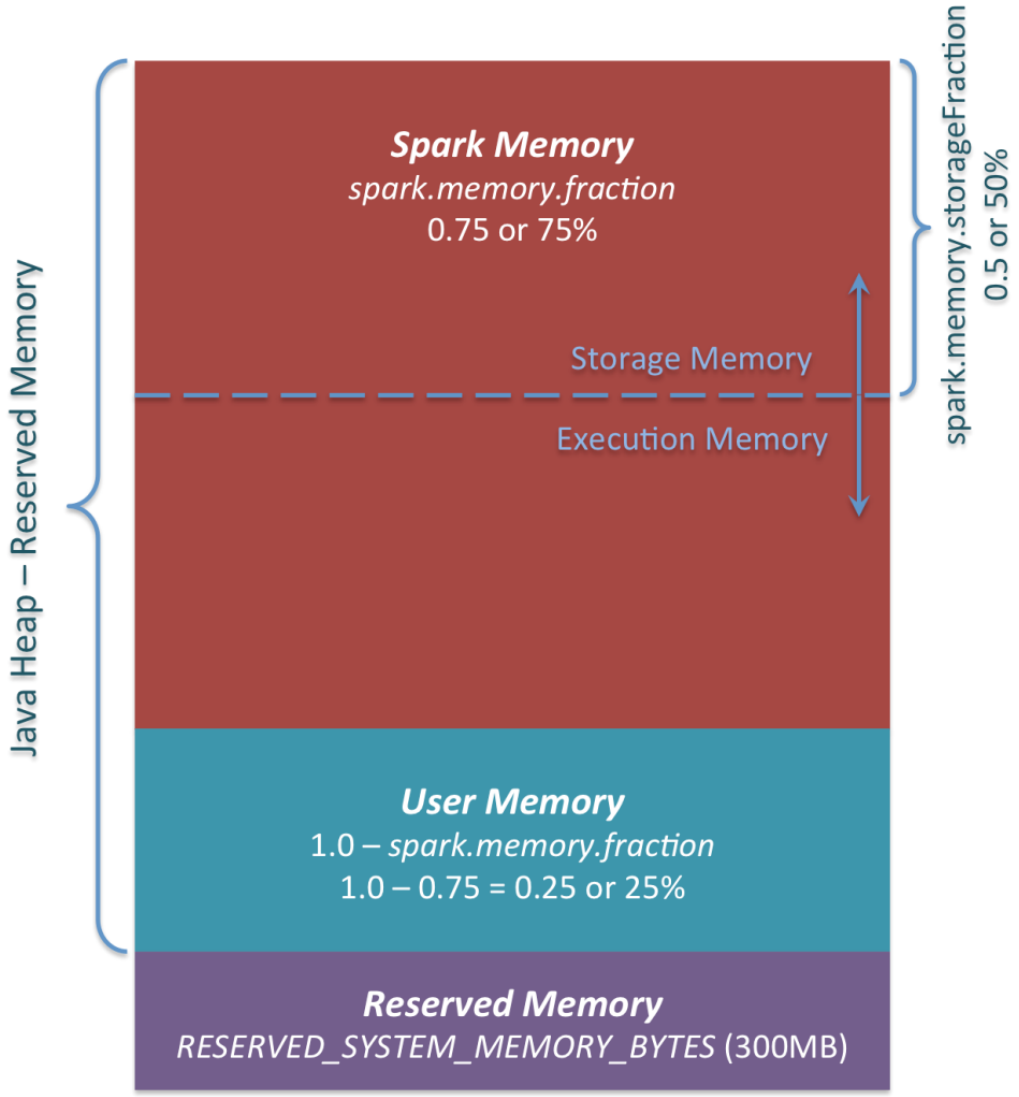
Terminologies (Contd.)

- **Worker Node :** Node that run the application program in cluster
- **Executor**
  - Process launched on a worker node, that runs the Tasks
  - Keep data in memory or disk storage
  - Cache Memory & Swap storage for RDD lineage
- **Job**
  - Consists multiple tasks, Created based on a Action
- **Stage :** Each Job is divided into a smaller set of tasks called Stages that is sequential and depend on each other
- **Task :** A unit of work that will be sent to executor.
- **Partitions:** Data unit that will be handled parallel, Same as Blocks in HDFS.





Memory Management in Spark



- **Execution Memory**
  - Memory used for shuffles, joins, sort and aggregations
- **Storage Memory**
  - storage of cached RDDs and broadcast variables
- **User Memory**
  - user data structures and internal metadata in Spark
- **Reserved memory**
  - memory needed for running executor itself and not strictly related to Spark

RUNNING SPARK ON CLUSTER

Hardware Requirements

CPU Cores

Spark scales well to tens of CPU cores per machine because it performs minimal sharing between threads.
You should likely provision at least 8-16 cores per machine. At max of 5 cores per executor.

Memory

Spark runs well with anywhere from **8 GB to hundreds of gigabytes of memory per machine**. In all cases, we recommend allocating only at **most 75% of the memory for Spark**; leave the rest for the operating system and buffer cache.

Local Disks

While Spark can perform a lot of its computation in memory, it still uses local disks to store data that doesn't fit in RAM, as well as to preserve intermediate output between stages.
We recommend having **4-8 disks per node, 2TB per drive**

Network

When the data is in memory, a lot of Spark applications are network-bound.
Using a **10 Gigabit** or higher network is the best way to make these applications faster. This is especially true for "distributed reduce" applications such as group-by, reduce-by, and SQL joins.

Cluster Resource Managers

Local
- Single JVM to run the whole process
- Spark-shell by default run on local mode

Spark Standalone (AMPLab, Spark default)
- Suitable for a lot of production workloads
- Only Spark workloads

YARN (Hadoop)
- Allows hierarchies of resources
- Kerberos integration
- Multiple workloads from different execution frameworks
- Hive, Pig, Spark, MapReduce, etc.,

Mesos (AMPLab)
- Similar to YARN, but allows elastic allocation to disparate execution frameworks
- Coarse-grained - Single, long-running Mesos tasks runs Spark mini tasks
- Fine-grained - New Mesos task for each Spark task, Higher overhead, not good for long-running Streaming Spark jobs

Starting the Spark Shell on a Cluster

- **Set the Spark Shell master to**
  - ***url*** – the URL of the cluster manager
  - **`local[*]`** – run with as many threads as cores (default)
  - **`local[n]`** – run locally with *n* worker threads
  - **`local`** – run locally without distributed processing
- **This configures the `SparkContext.master` property**

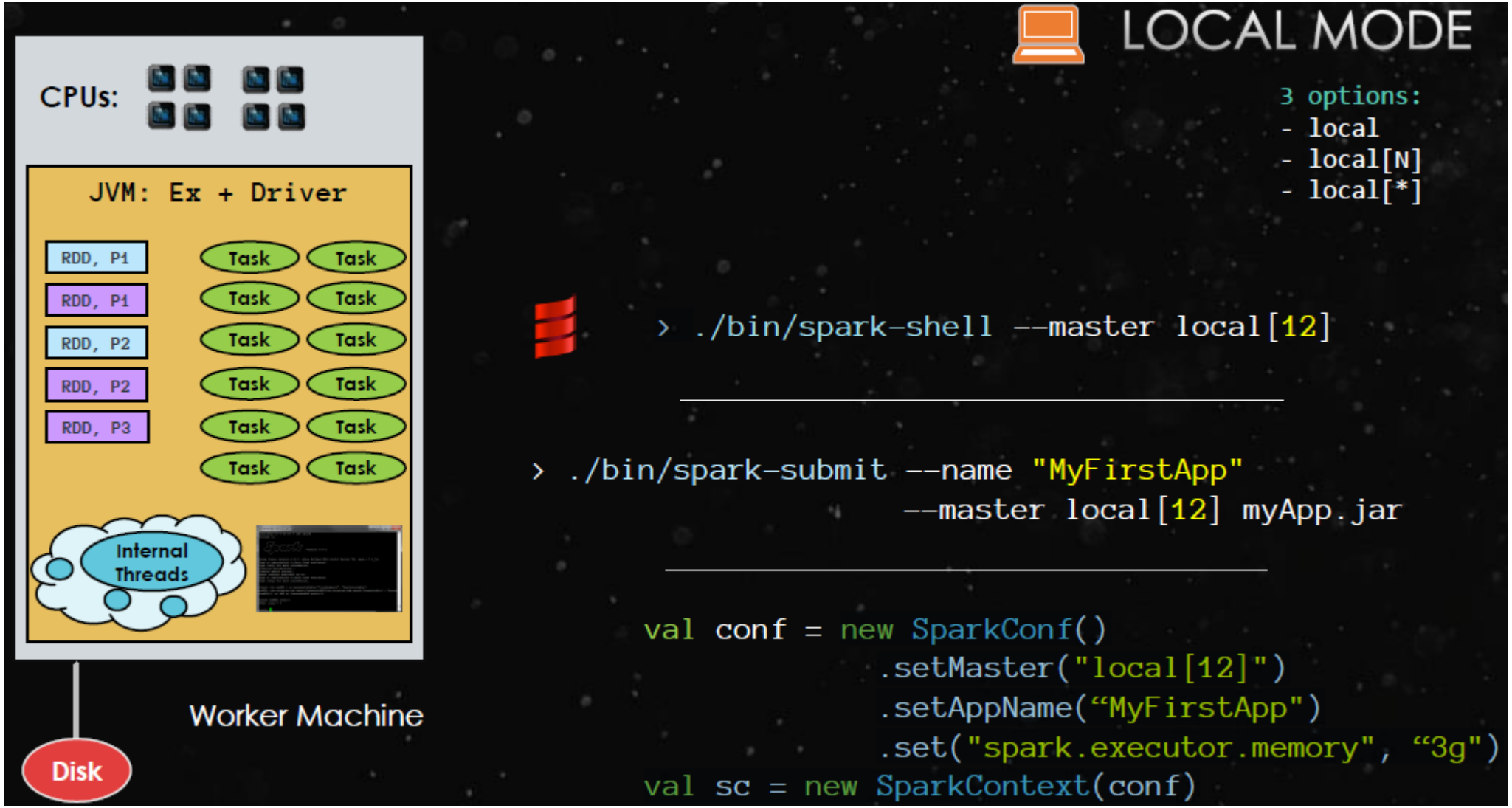
Python

```
$ MASTER=spark://masternode:7077 pyspark
```

Scala

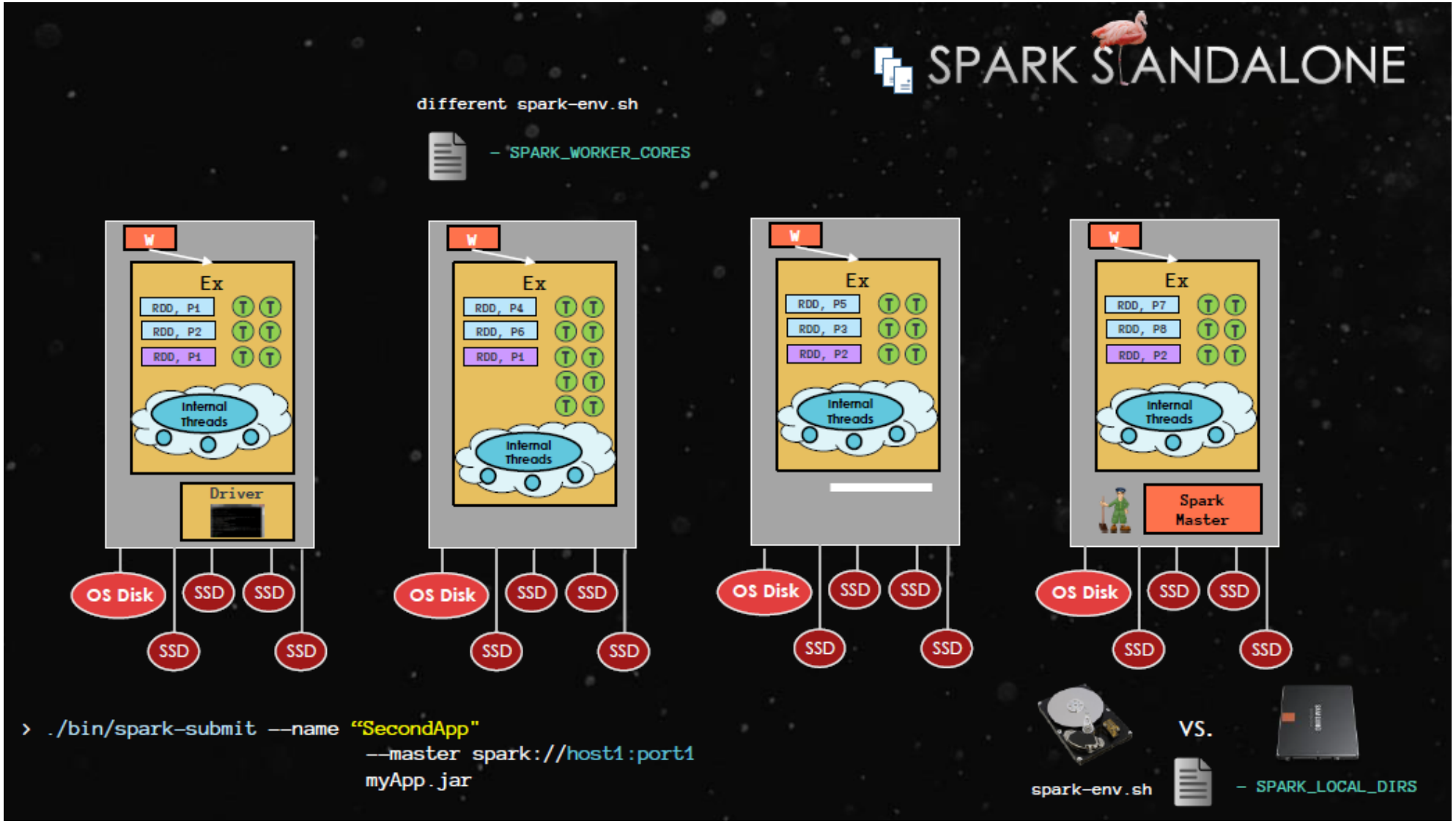
```
$ spark-shell --master spark://masternode:7077
```

Local Mode

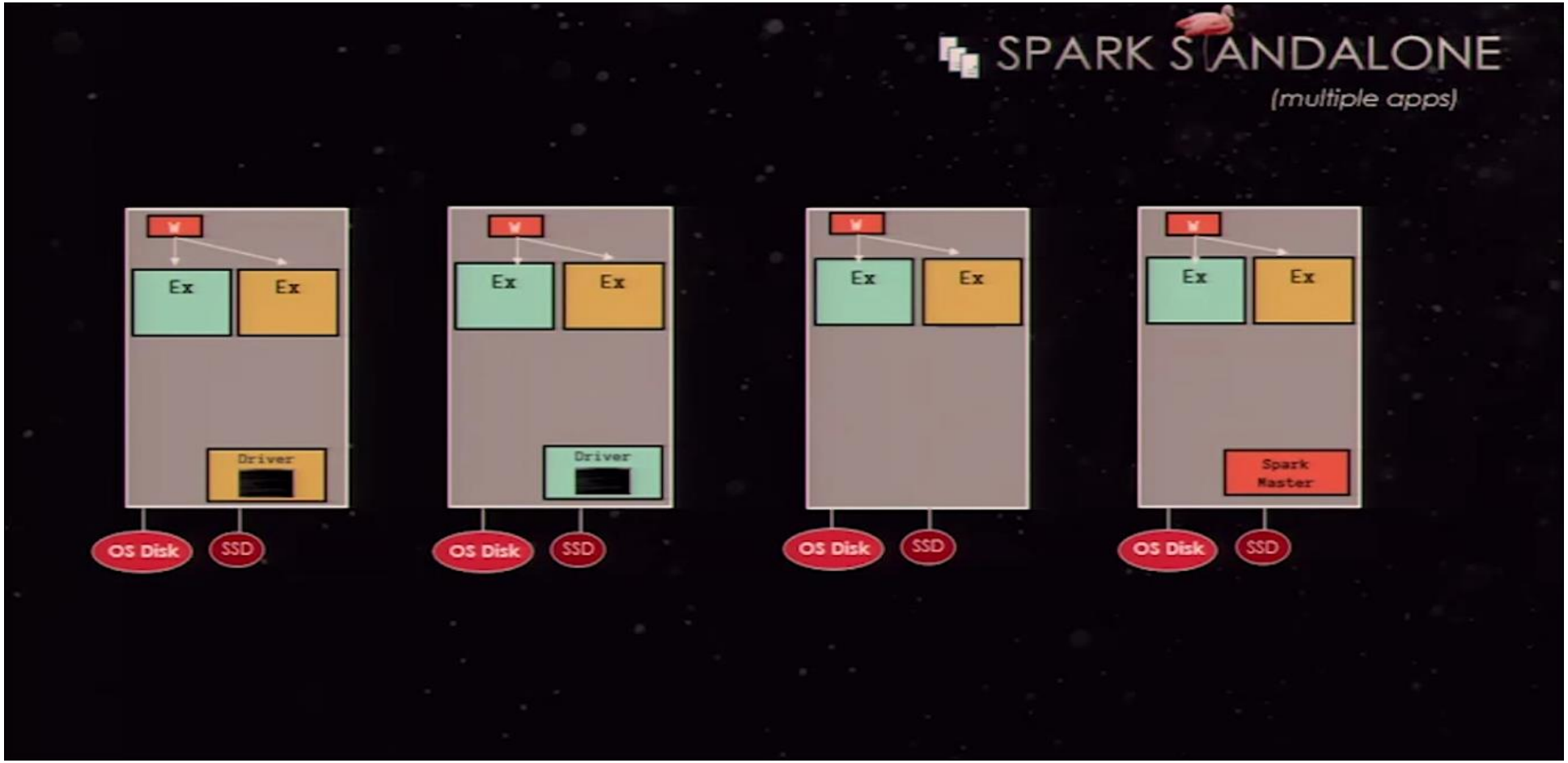

CPUs:
JVM: Ex + Driver
RDD, P1
RDD, P1
RDD, P2
RDD, P2
RDD, P3
Task
Internal Threads
Worker Machine
Disk
LOCAL MODE
3 options:
- local
- local[N]
- local[*]
› ./bin/spark-shell --master local[12]
› ./bin/spark-submit --name "MyFirstApp"
--master local[12] myApp.jar
val conf = new SparkConf()
.setMaster("local[12]")
.setAppName("MyFirstApp")
.set("spark.executor.memory", "3g")
val sc = new SparkContext(conf)

Standalone Mode

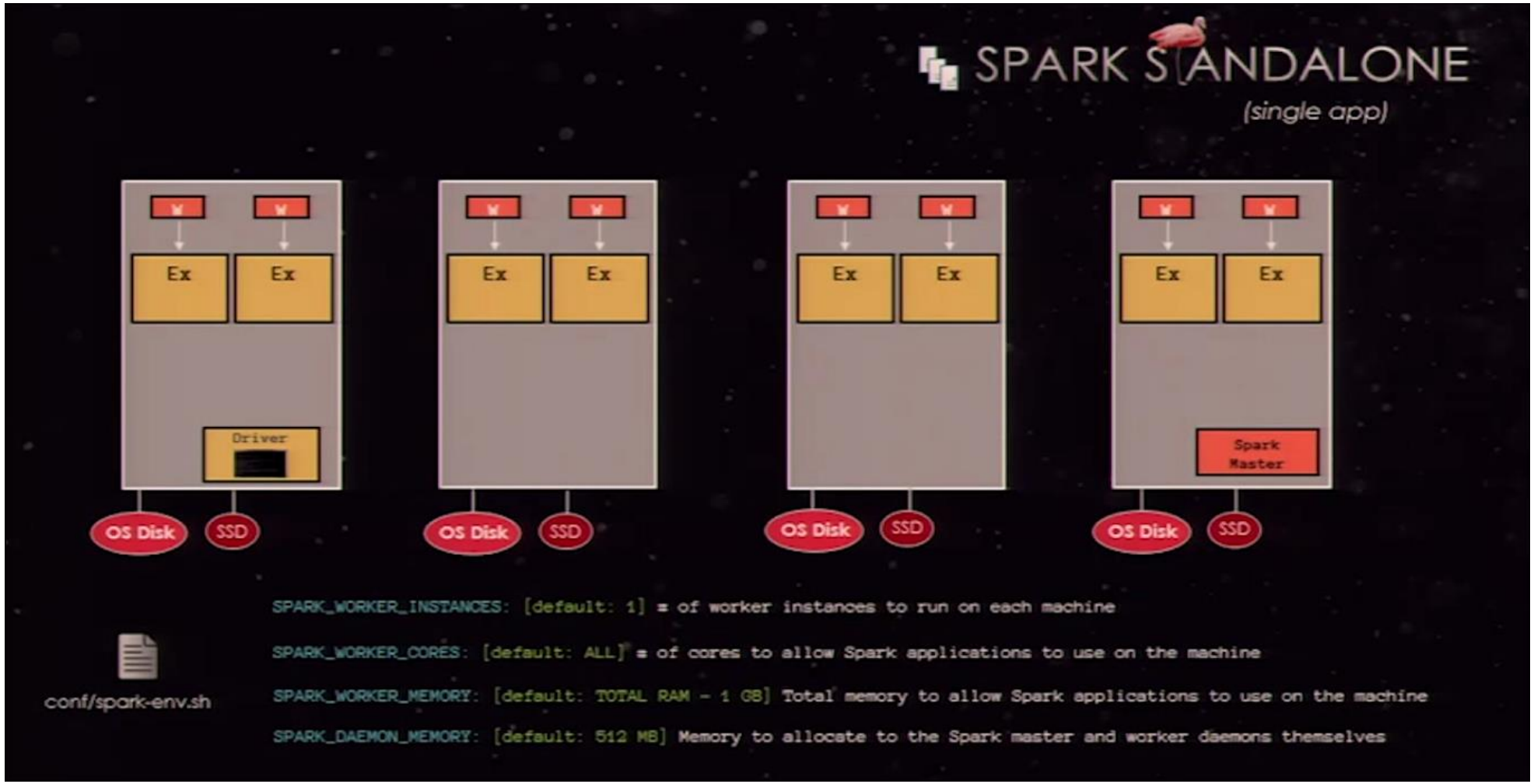
SPARK_WORKER_INSTANCES: [default: 1] # of worker instances to run on each machine
SPARK_WORKER_CORES: [default: ALL] # of cores to allow Spark applications to use on the machine
SPARK_WORKER_MEMORY: [default: TOTAL RAM – 1 GB] Total memory to allow Spark applications to use on the machine
SPARK_DAEMON_MEMORY: [default: 512 MB] Memory to allocate to the Spark master and worker daemons themselves



Standalone Mode..Cont.

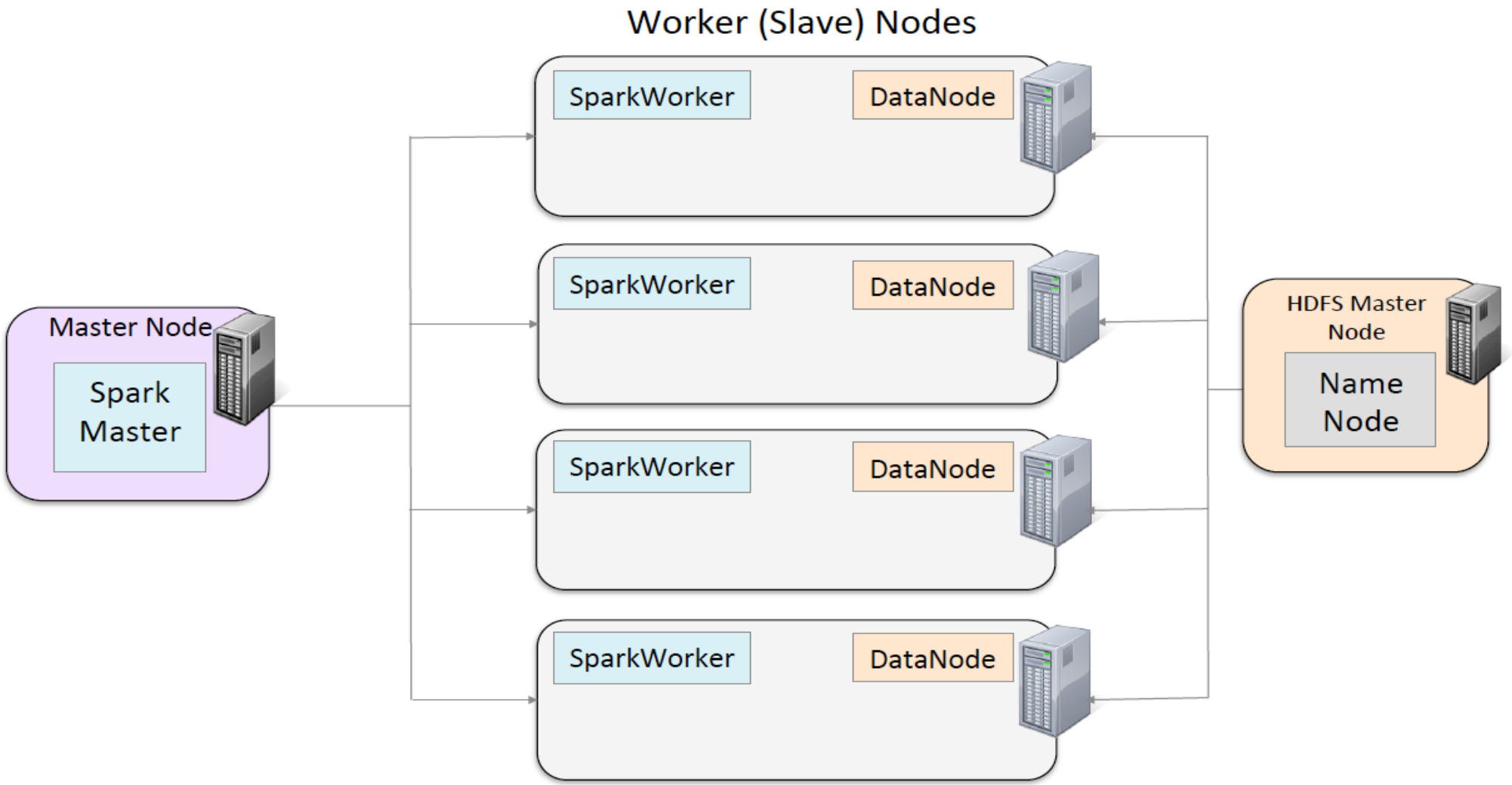


Standalone..Cont.

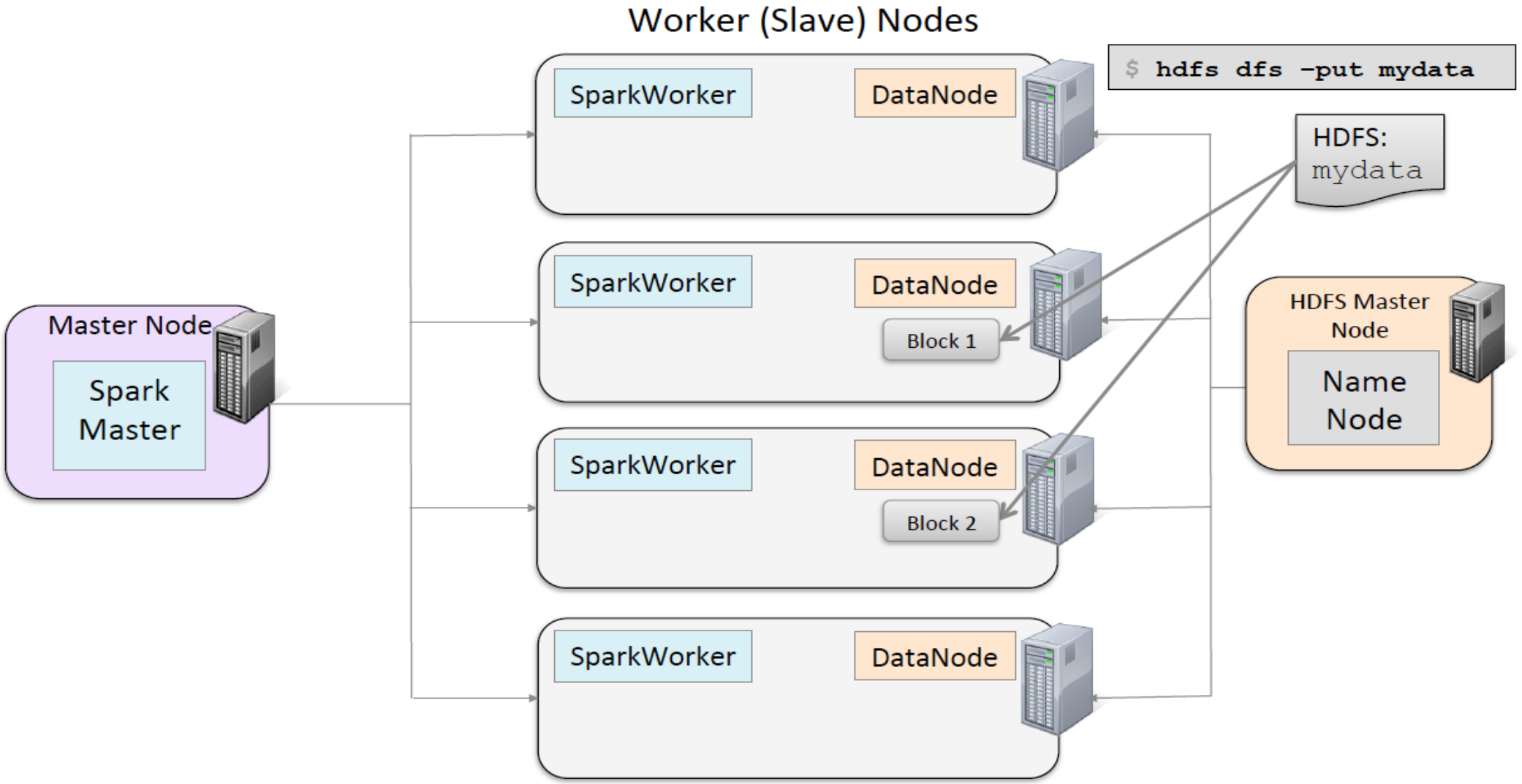


SPARK_WORKER_CORES – for instance there are 12 cores, 2 executors are there, worker allocates 6 cores for each executor
SPARK_WORKER_MEMORY – for instance there are 20 gigs RAM, 10 gigs of RAM allocated to each executor

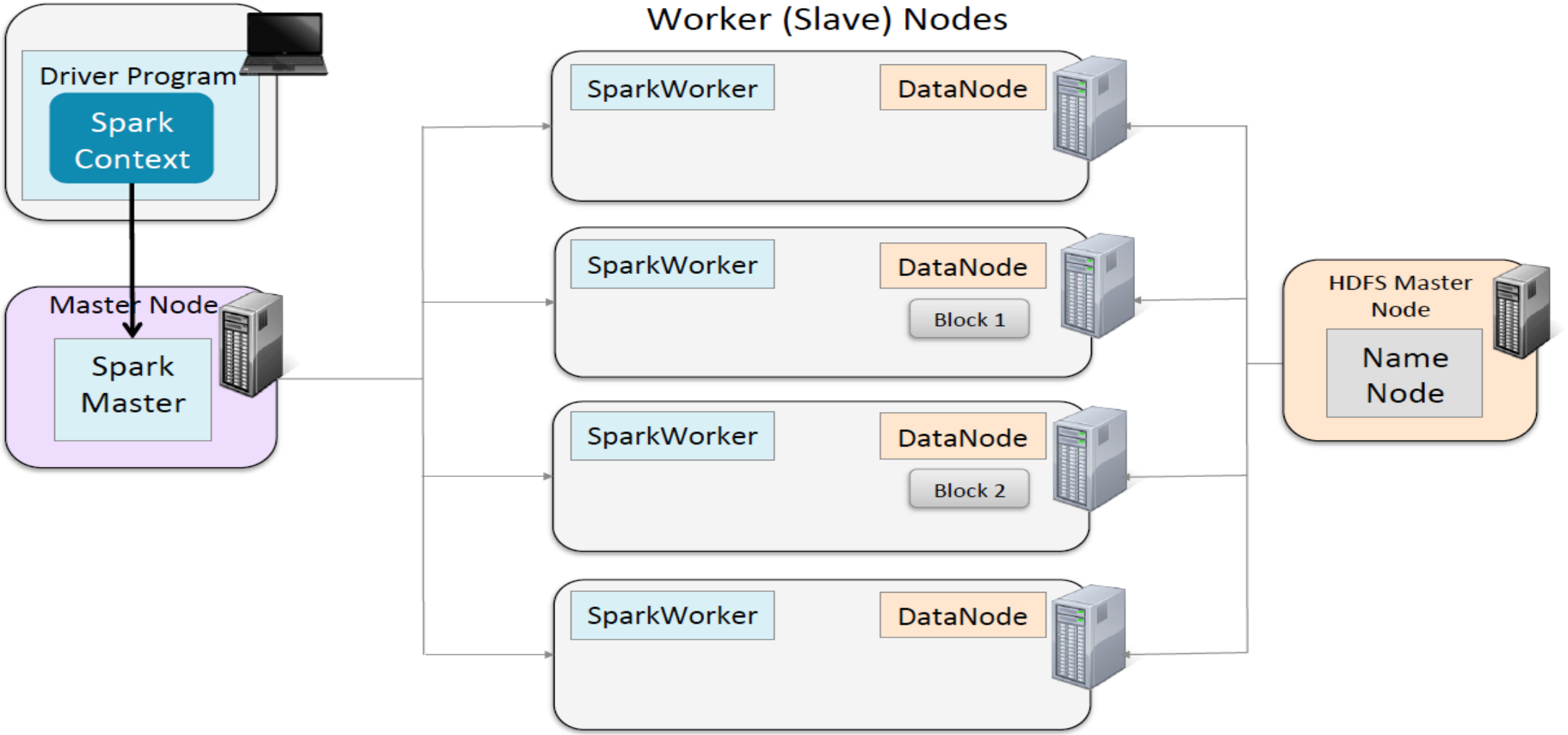
Running Spark on a Standalone Cluster (1)



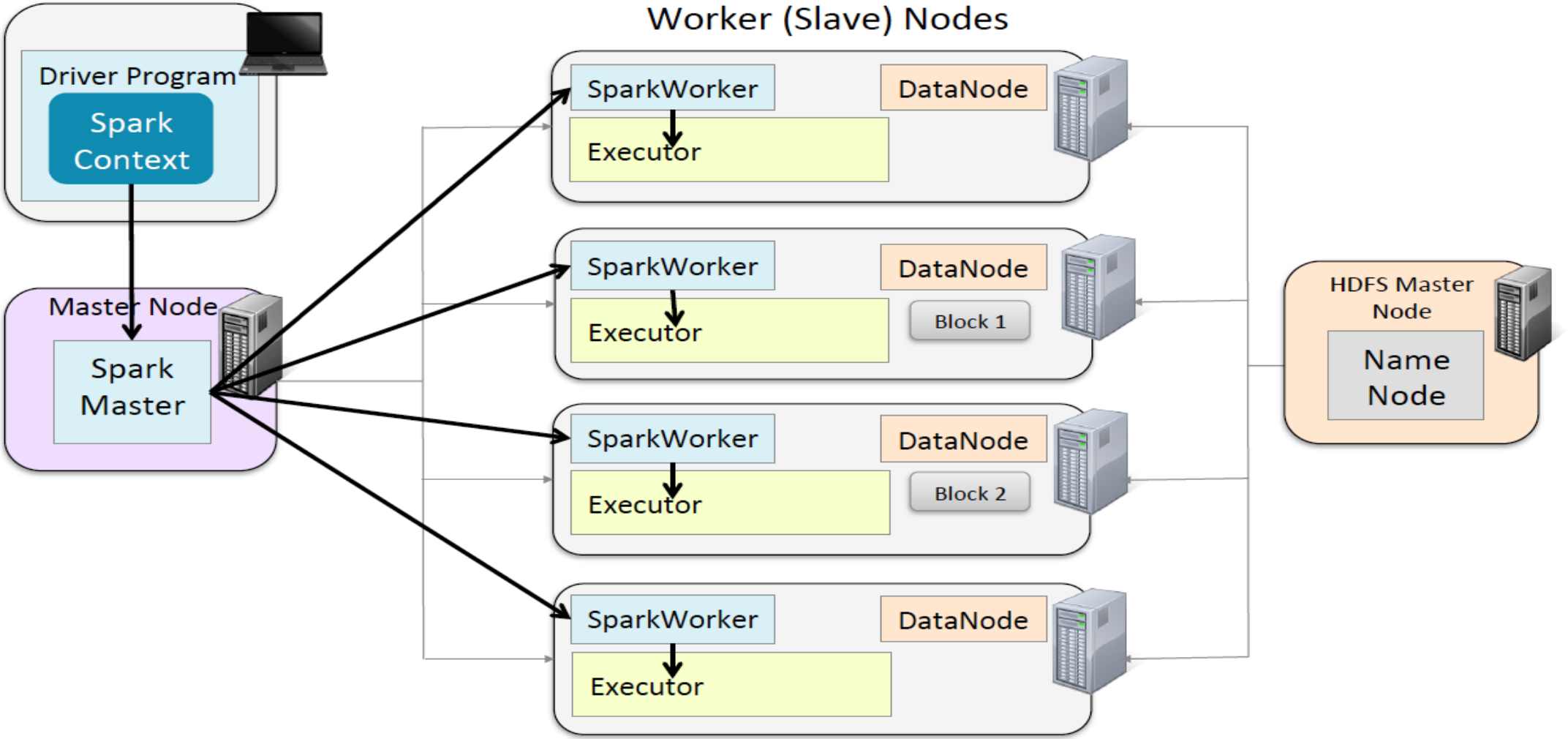
Running Spark on a Standalone Cluster (2)



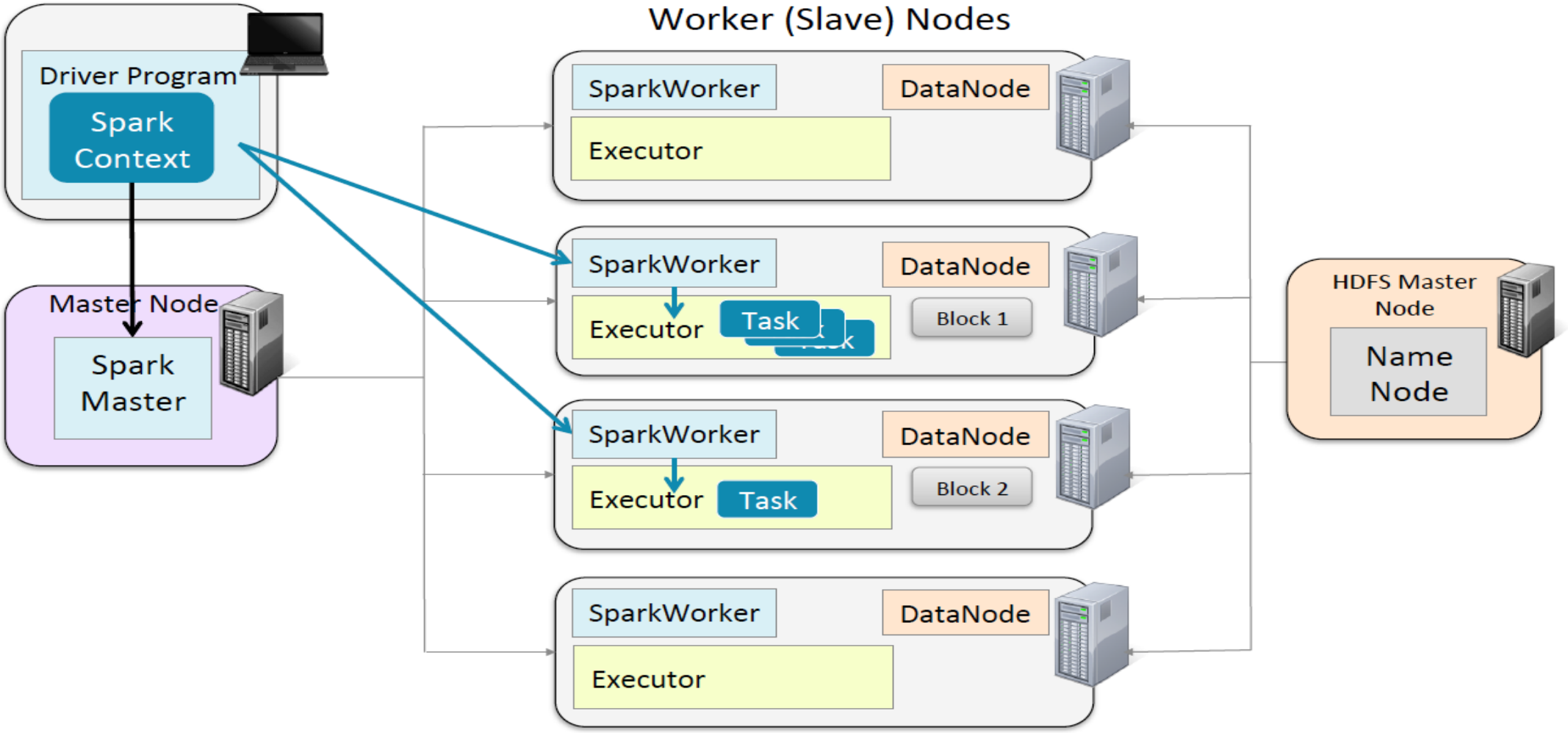
Running Spark on a Standalone Cluster (3)



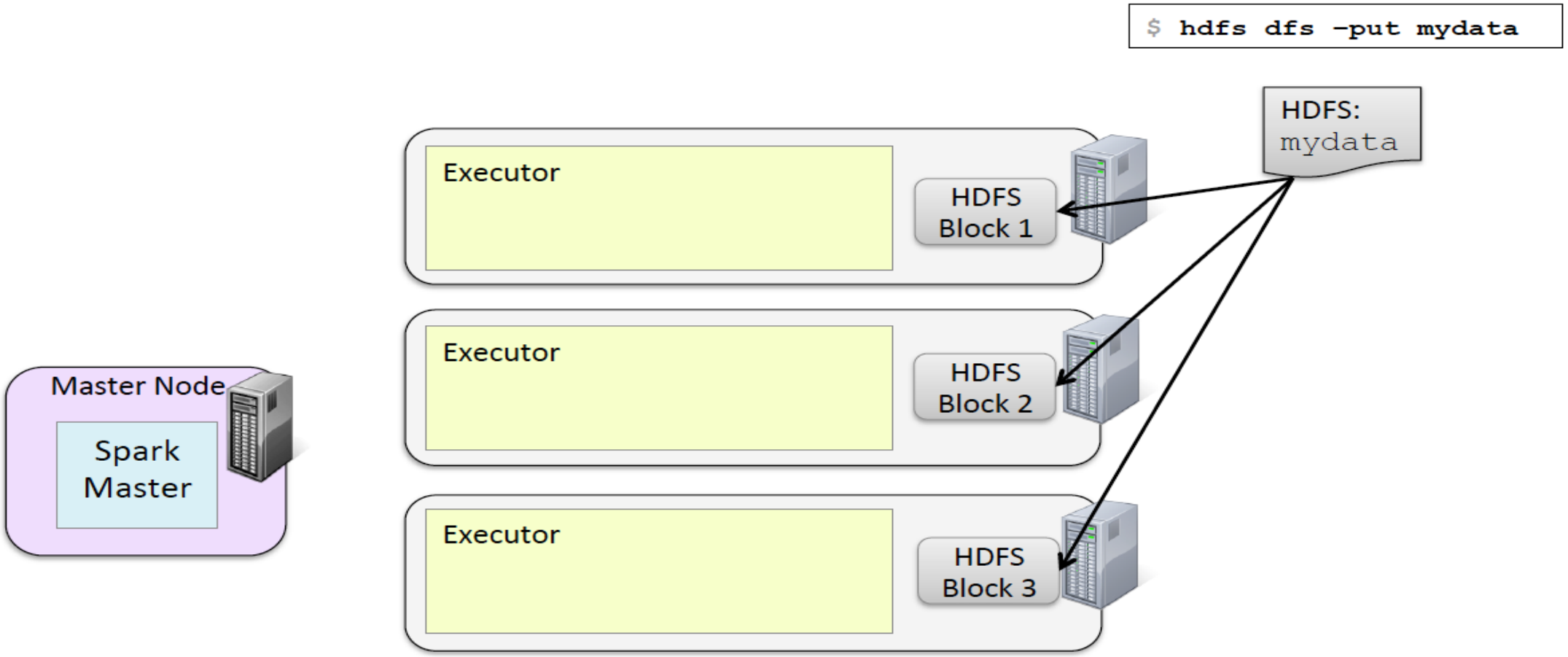
Running Spark on a Standalone Cluster (4)



Running Spark on a Standalone Cluster (5)



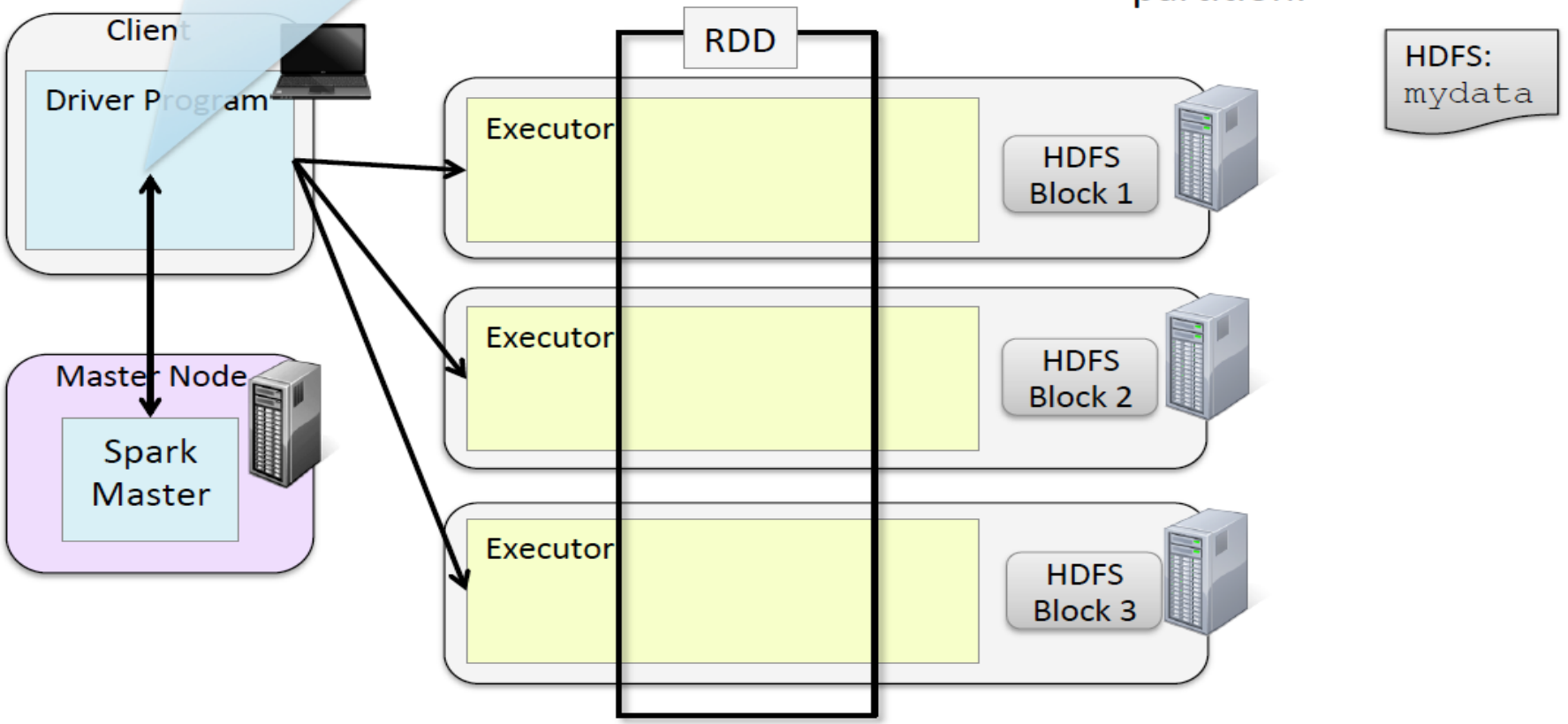
HDFS and Data Locality (1)



HDFS and Data Locality (2)

```
sc.textFile("hdfs://…mydata…").collect()
```

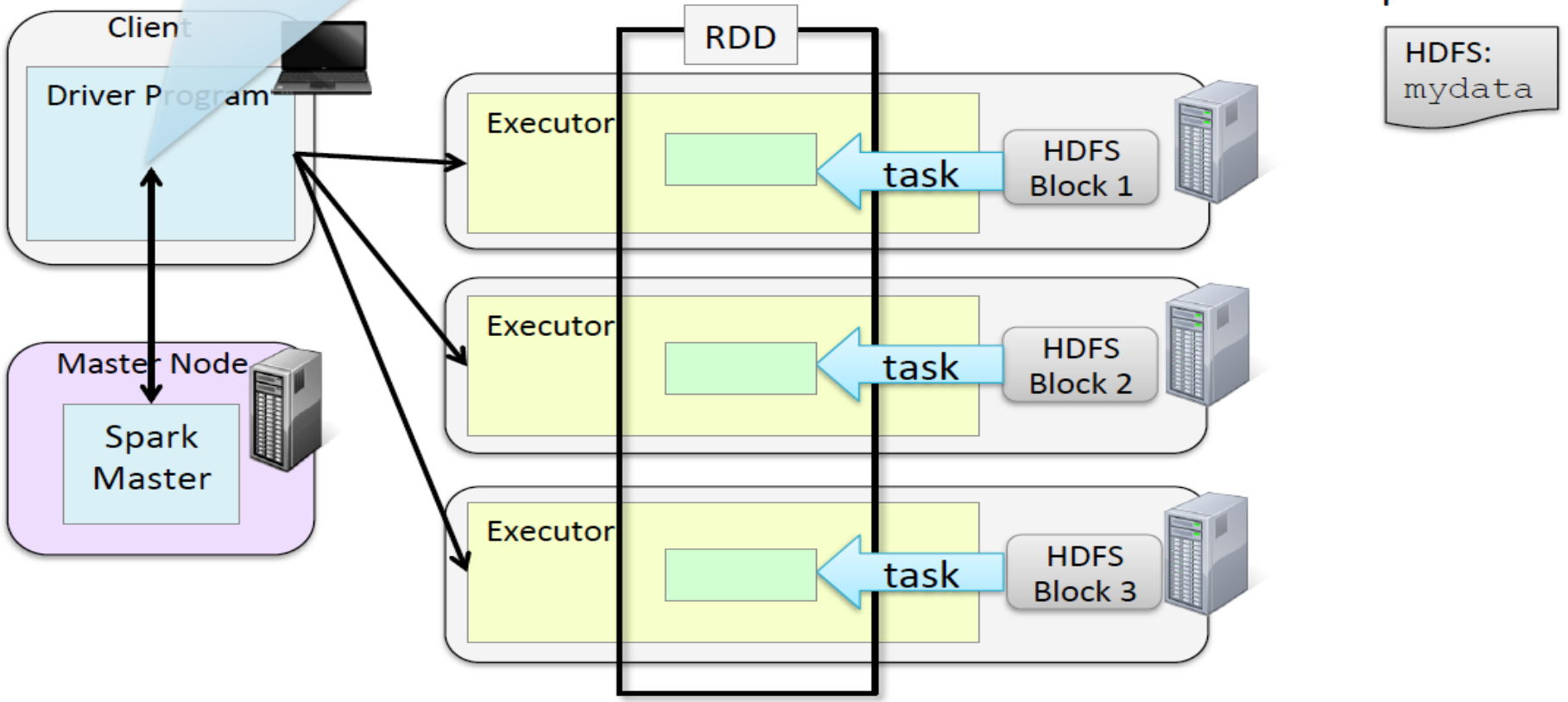
By default, Spark partitions file-based RDDs by block. Each block loads into a single partition.



HDFS and Data Locality (3)

```
sc.textFile("hdfs://…mydata…").collect()
```

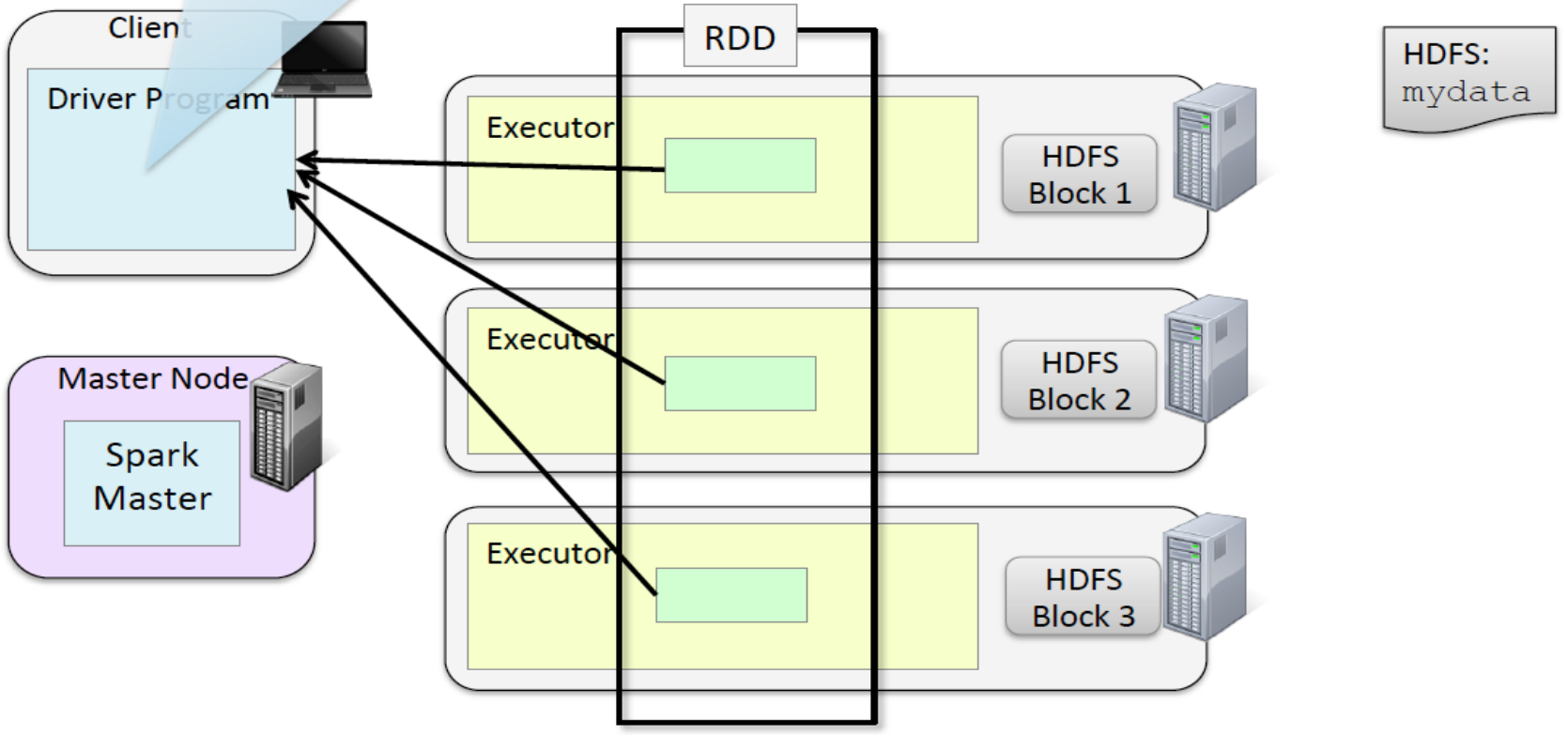
An action triggers execution: tasks on executors load data from blocks into partitions



HDFS and Data Locality (4)

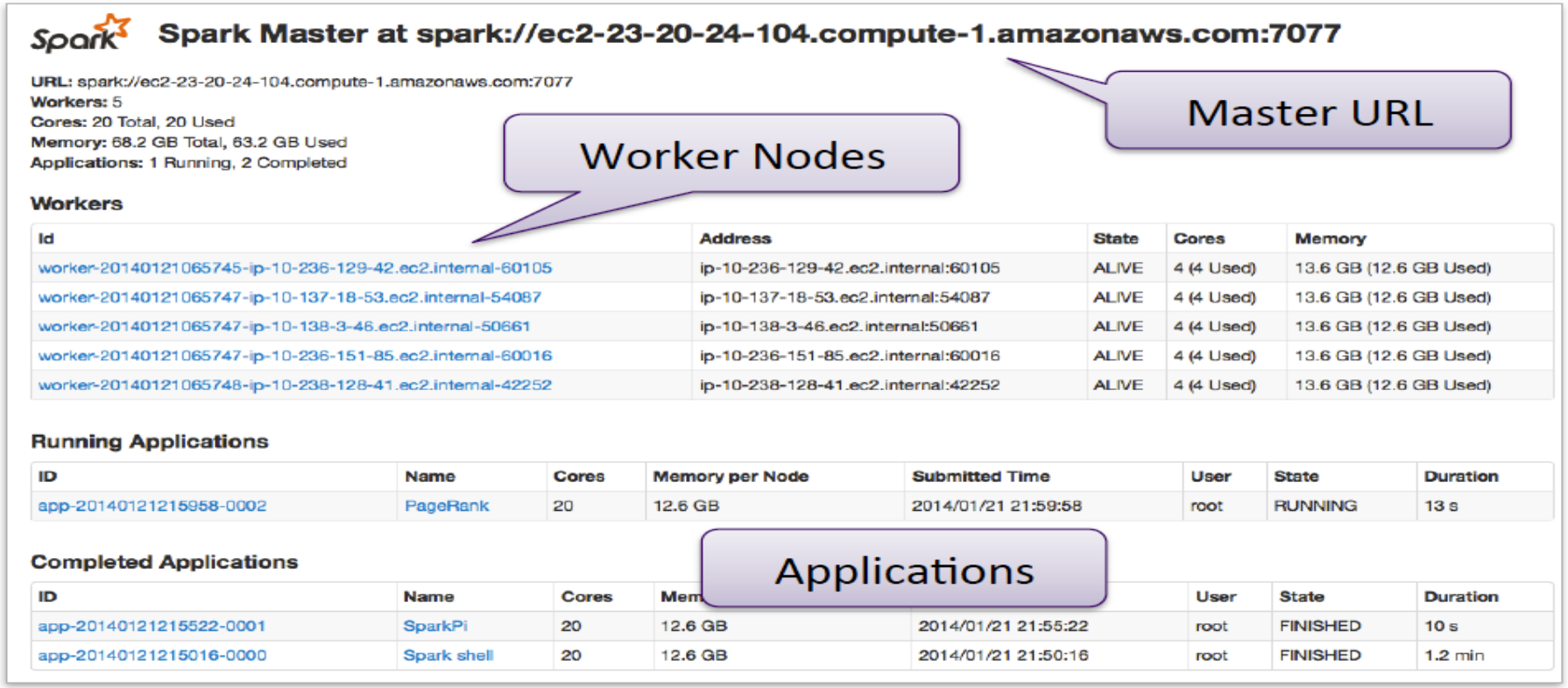
```
sc.textFile("hdfs://…mydata…").collect()
```

Data is distributed across executors until an action returns a value to the driver

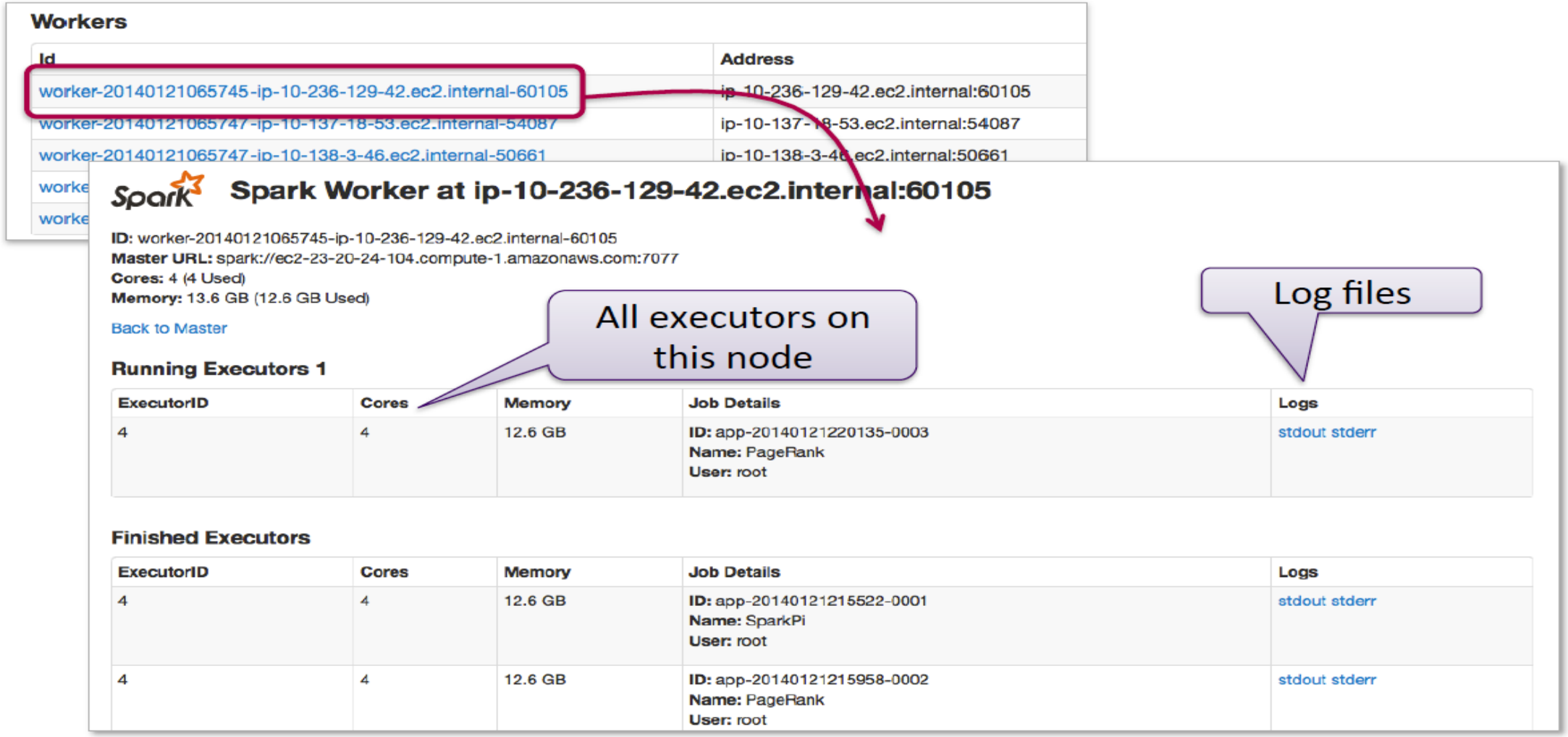


Spark Standalone Web UI

- **Spark Standalone clusters offer a Web UI to monitor the cluster**
  - **`http://masternode:uiport`**
    - e.g., in our class environment, **`http://localhost:18080`**



Spark Standalone Web UI: Worker Detail



Workers

Id	Address
worker-20140121065745-ip-10-236-129-42.ec2.internal-60105	ip-10-236-129-42.ec2.internal:60105
worker-20140121065747-ip-10-137-18-53.ec2.internal-54087	ip-10-137-18-53.ec2.internal:54087
worker-20140121065747-ip-10-138-3-46.ec2.internal-50661	ip-10-138-3-46.ec2.internal:50661
worke	
worke	

Spark Worker at ip-10-236-129-42.ec2.internal:60105

ID: worker-20140121065745-ip-10-236-129-42.ec2.internal-60105
Master URL: spark://ec2-23-20-24-104.compute-1.amazonaws.com:7077
Cores: 4 (4 Used)
Memory: 13.6 GB (12.6 GB Used)

Back to Master

All executors on this node

Log files

Running Executors 1

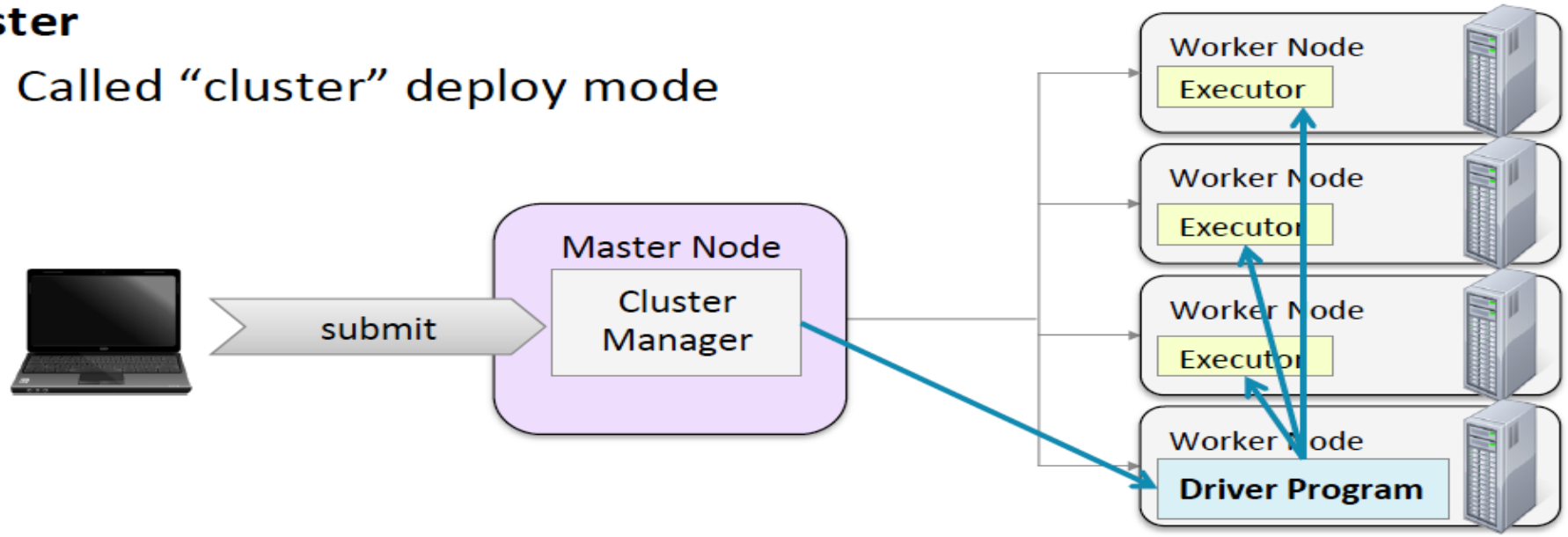
ExecutorID	Cores	Memory	Job Details	Logs
4	4	12.6 GB	**ID:** app-20140121220135-0003 **Name:** PageRank **User:** root	stdout stderr

Finished Executors

ExecutorID	Cores	Memory	Job Details	Logs
4	4	12.6 GB	**ID:** app-20140121215522-0001 **Name:** SparkPi **User:** root	stdout stderr
4	4	12.6 GB	**ID:** app-20140121215958-0002 **Name:** PageRank **User:** root	stdout stderr

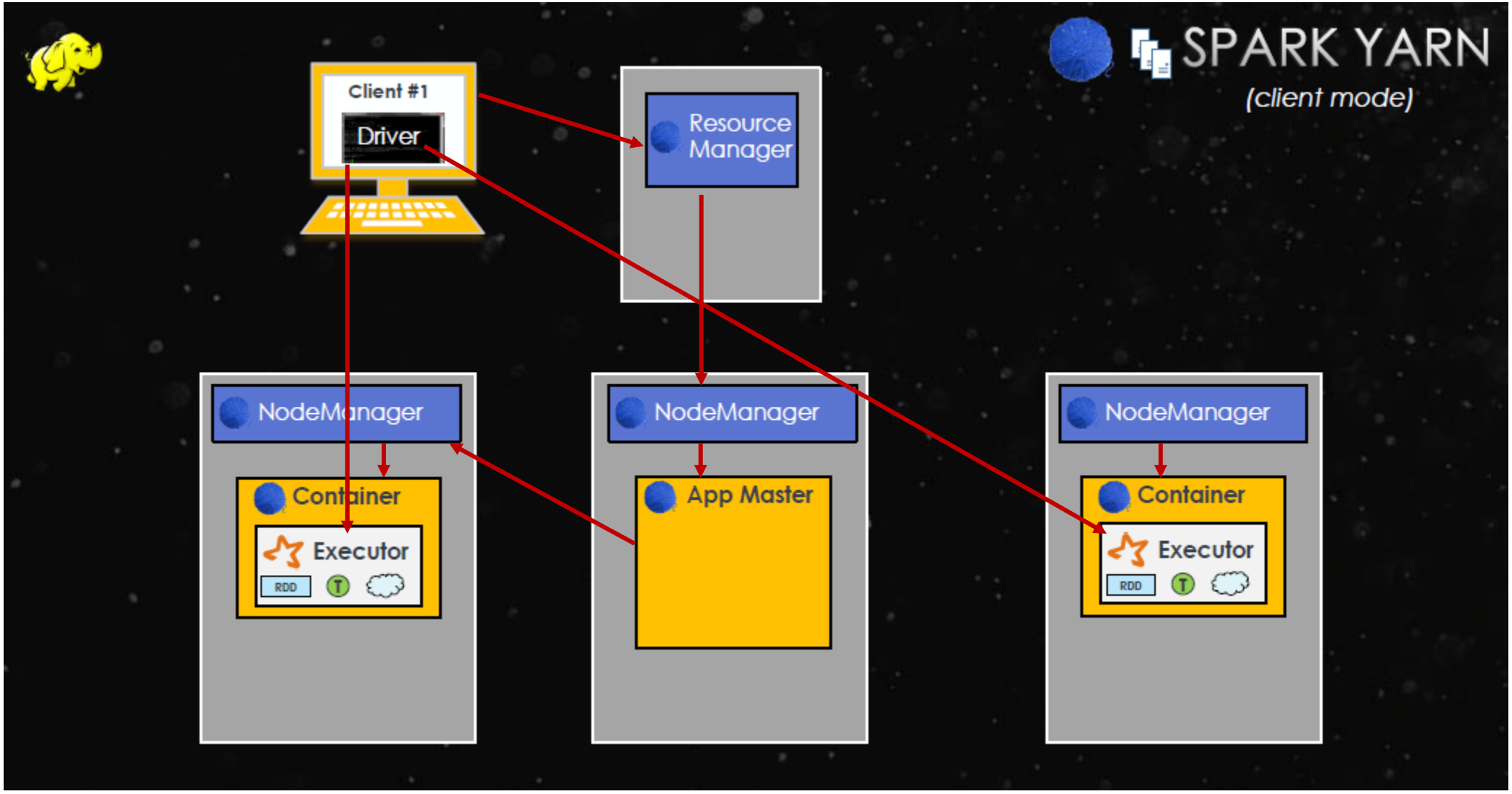
YARN Modes

- **By default, the driver program runs outside the cluster**
  - Called “client” deploy mode
  - Most common
  - Required for interactive use (e.g., the Spark Shell)
- **It is also possible to run the driver program on a worker node in the cluster**
  - Called “cluster” deploy mode

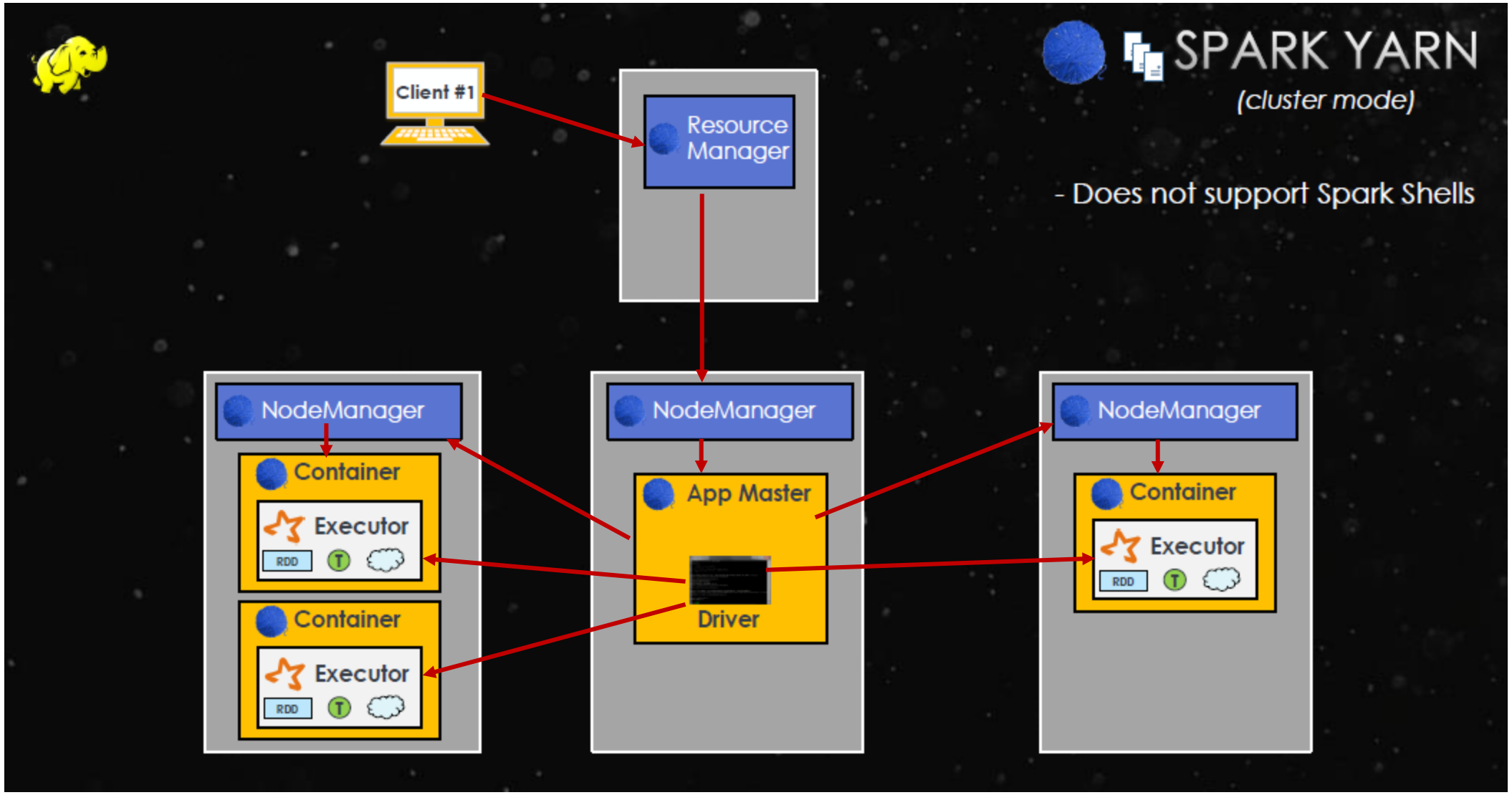


YARN – Client Mode

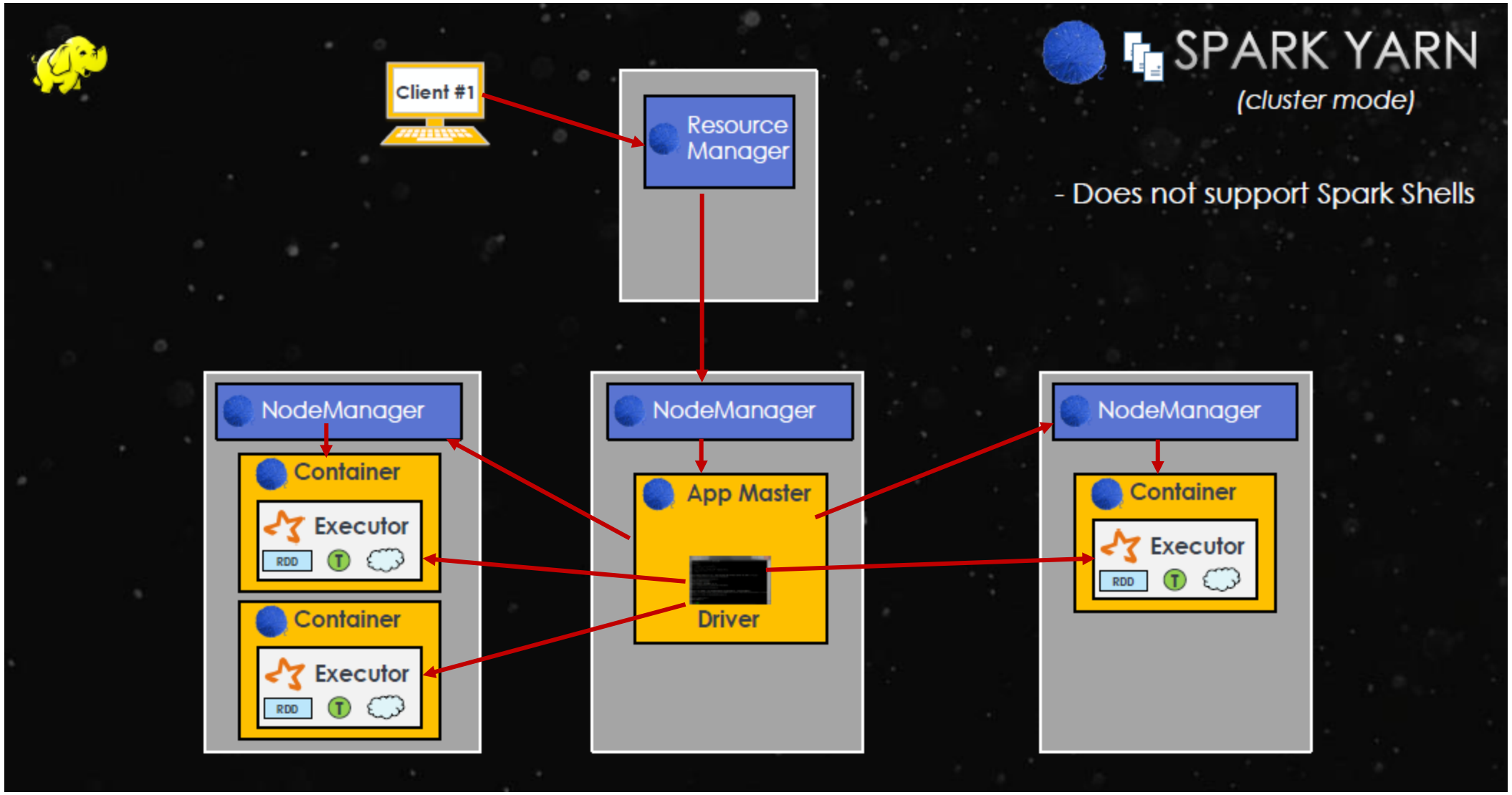
--num-executors: controls how many executors will be allocated
--executor-memory: RAM for each executor
--executor-cores: CPU cores for each executor



YARN – Cluster Mode



YARN – Cluster Mode



Spark Submit - Different Clusters

```
# Run application locally on 8 cores
./bin/spark-submit \
  --class org.apache.spark.examples.SparkPi \
  --master local[8] \
  /path/to/examples.jar \
  100
```

```
# Run on a Spark standalone cluster in client deploy mode
./bin/spark-submit \
  --class org.apache.spark.examples.SparkPi \
  --master spark://207.184.161.138:7077 \
  --executor-memory 20G \
  --total-executor-cores 100 \
  /path/to/examples.jar \
  1000
```

```
# Run on a Spark standalone cluster in cluster deploy
mode with supervise
./bin/spark-submit \
  --class org.apache.spark.examples.SparkPi \
  --master spark://207.184.161.138:7077 \
  --deploy-mode cluster \
  --supervise \
  --executor-memory 20G \
  --total-executor-cores 100 \
  /path/to/examples.jar \
  1000
```

```
# Run on a YARN cluster
./bin/spark-submit \
  --class org.apache.spark.examples.SparkPi \
  --master yarn \
  --deploy-mode cluster \  # can be client for client mode
  --executor-memory  20G \
  --num-executors  50 \
  /path/to/examples.jar \
  1000
```

```
# Run a Python application on a Spark standalone cluster
./bin/spark-submit \
  --master spark://207.184.161.138:7077 \
  examples/src/main/python/pi.py \
  1000
```

```
# Run on a Mesos cluster in cluster deploy mode with supervise
./bin/spark-submit \
  --class org.apache.spark.examples.SparkPi \
  --master mesos://207.184.161.138:7077 \
  --deploy-mode cluster \
  --supervise \
  --executor-memory 20G \
  --total-executor-cores 100 \
  http://path/to/examples.jar \
  1000
```

SPARK DISTRIBUTION SETUP AND CONFIGURATIONS (WITH HANDS ON SESSIONS)

DEMO