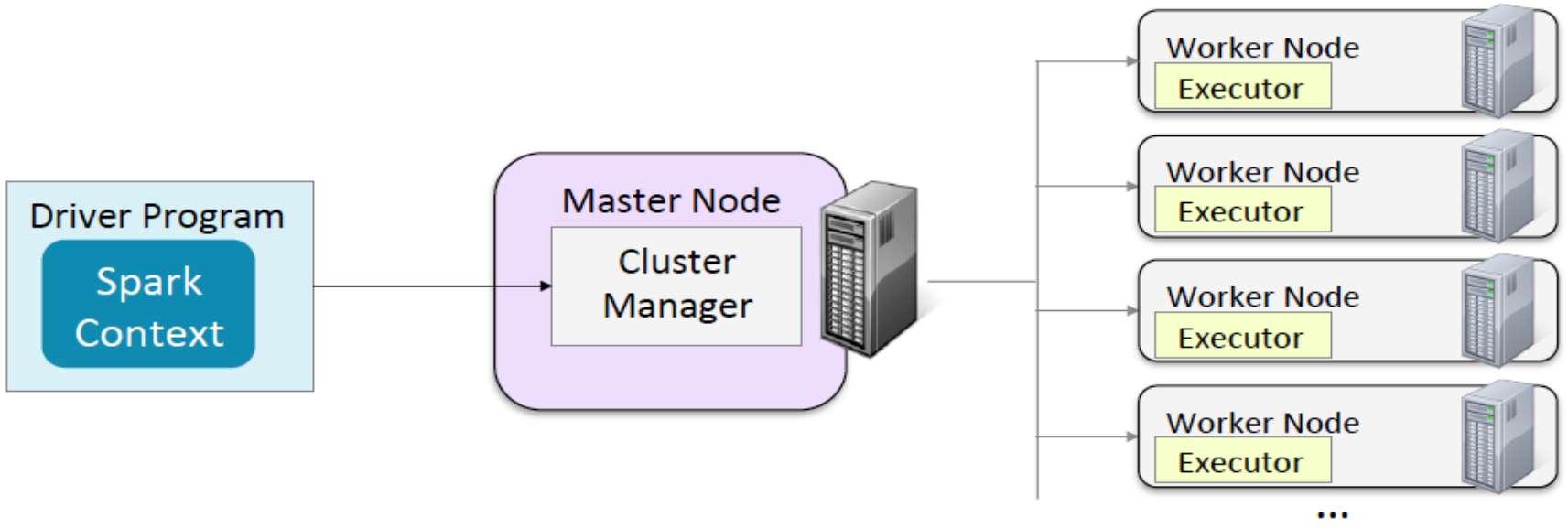
SPARK ESSENTIALS

The Spark Driver Program

- **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



Spark Essentials : SparkContext

First thing that a Spark program does is create a `SparkContext` object, which tells Spark how to access a cluster

In the shell for either Scala or Python, this is the `sc` variable, which is created automatically

Other programs must use a constructor to instantiate a new `SparkContext`

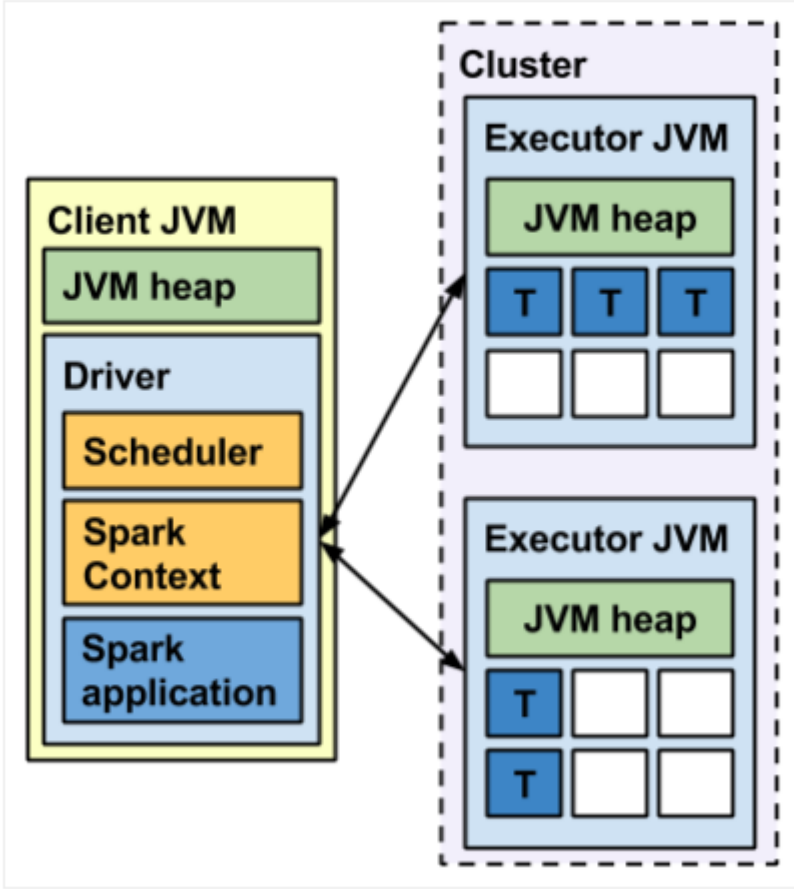
Then in turn `SparkContext` gets used to create other variables

Scala:

```
scala> sc
res: spark.SparkContext = spark.SparkContext@470d1f30
```

Python:

```
>>> sc
<pyspark.context.SparkContext object at 0x7f7570783350>
```

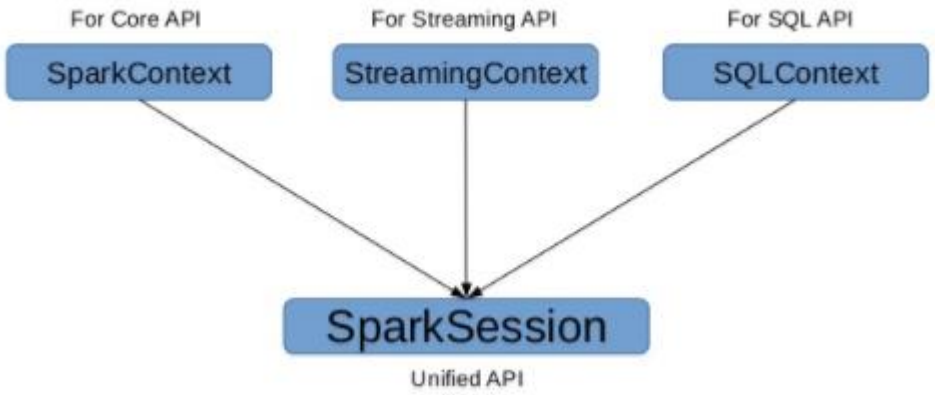


SparkSession – New entry point of Spark 2.0

SparkSession is the new starting point for Spark SQL-based applications

It encapsulates SparkContext, SparkConf, SQLContext and HiveContext

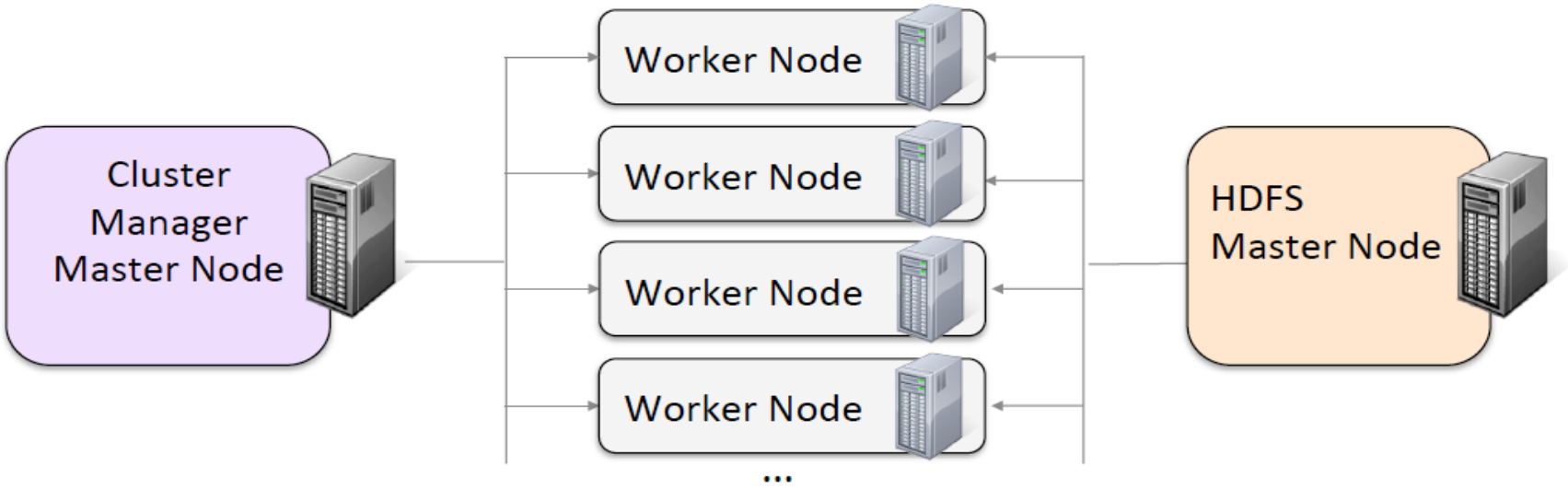
SparkSession contains all the features that were present in them.




```
val sparkSession = SparkSession.builder.
      master("local")
      .appName("spark session example")
      .enableHiveSupport()
      .getOrCreate()
```

Spark Cluster Terminology

- **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

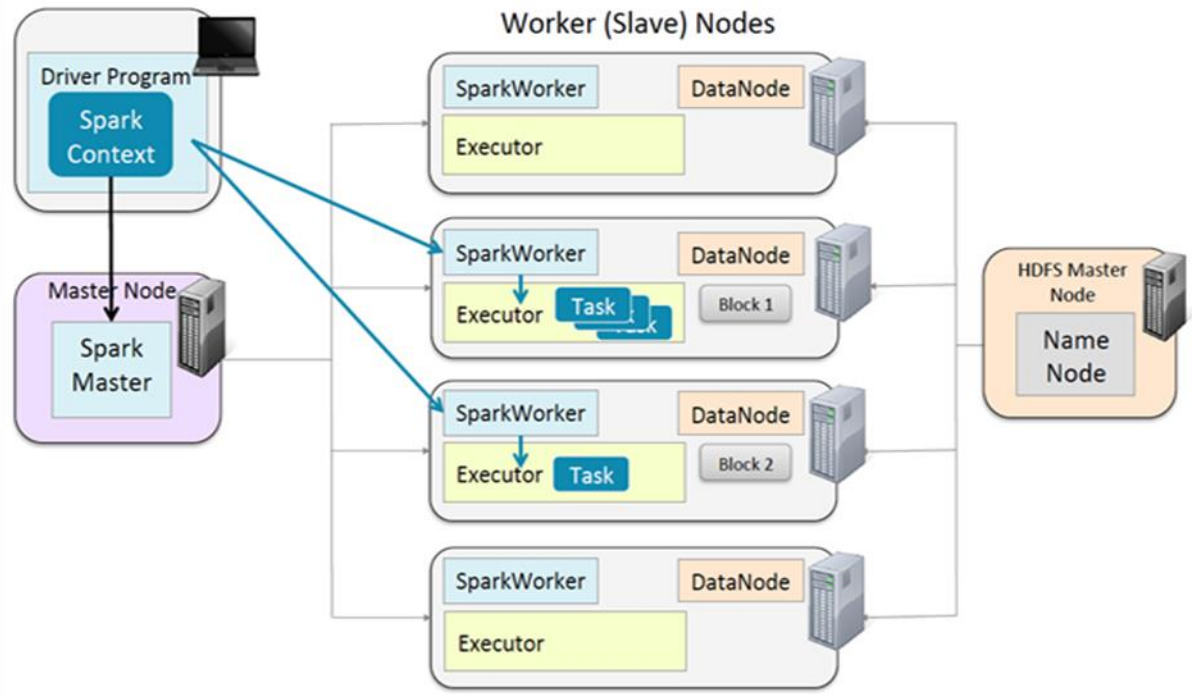


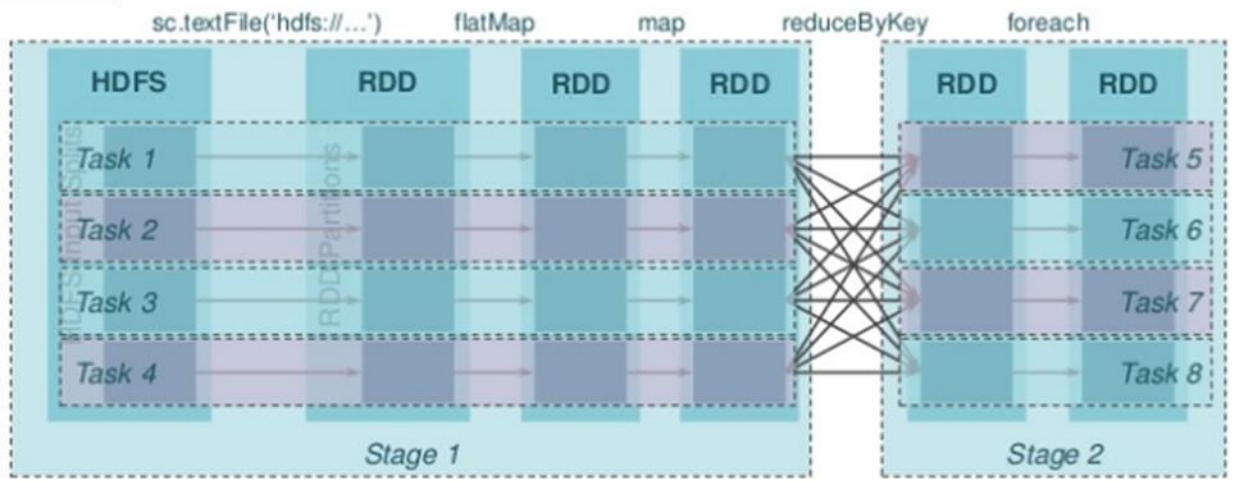
Terminologies

- **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

Terminology (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.





RUNNING SPARK ON CLUSTER

Cluster Resource Managers

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**

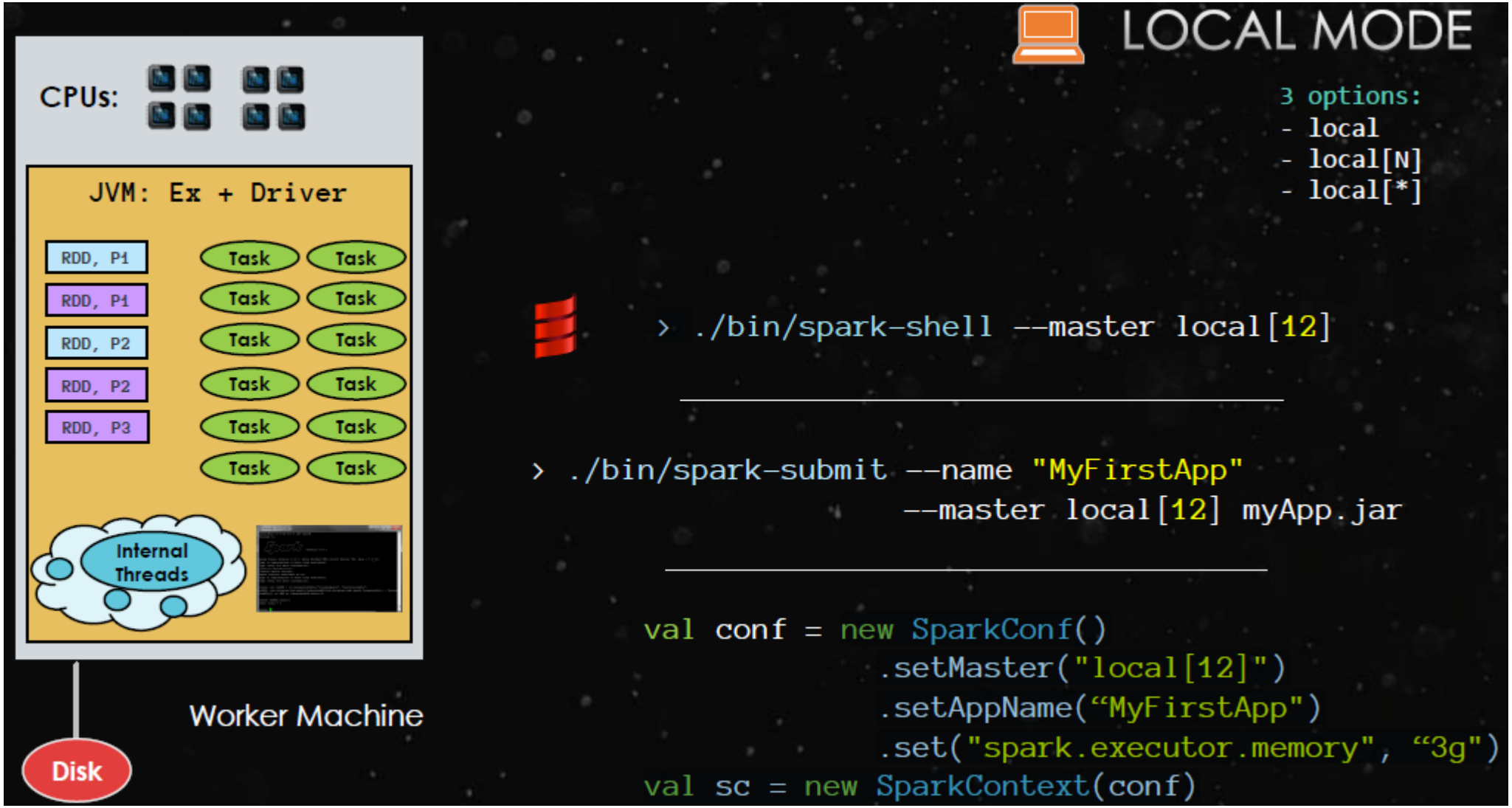
Python

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

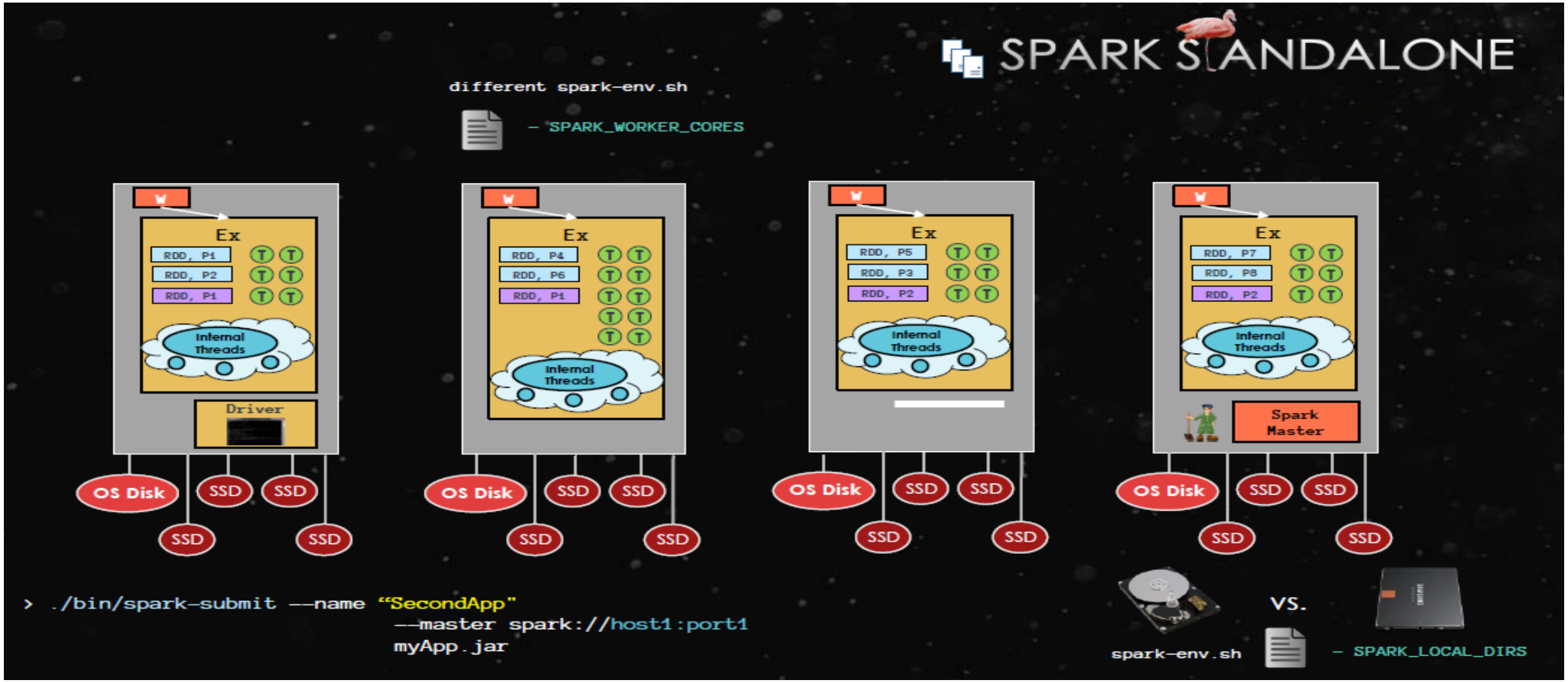
Scala

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

Local Mode

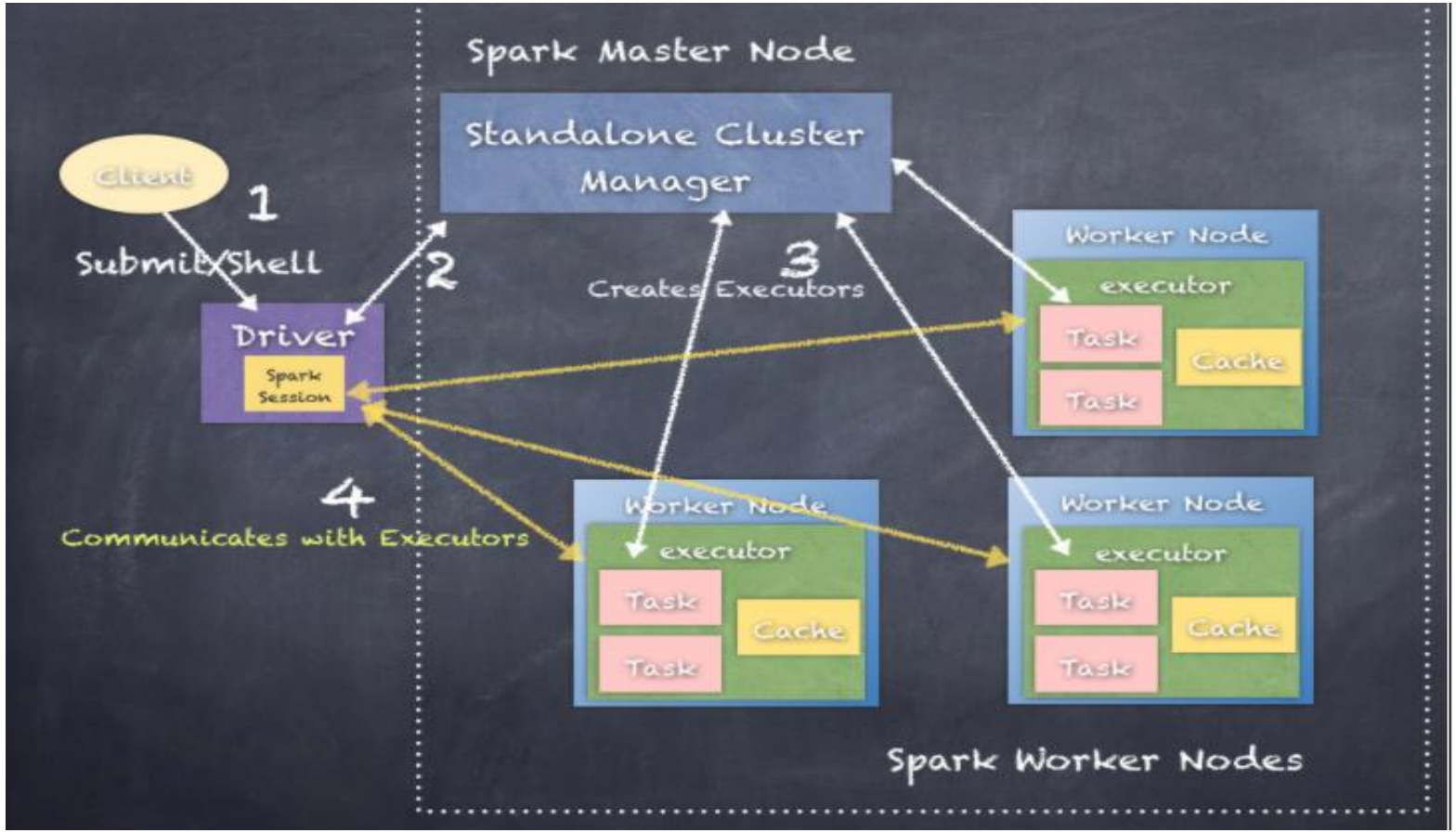

CPUs:
JVM: Ex + Driver
RDD, P1
RDD, P1
RDD, P2
RDD, P2
RDD, P3
Task
Internal Threads
Worker Machine
Disk
LOCAL MODE
3 options:
- local
- local[N]
- local[*]
› ./bin/spark-shell --master local[12]
› ./bin/spark-submit --name "MyFirstApp"
--master local[12] myApp.jar
val conf = new SparkConf()
.setMaster("local[12]")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

Standalone Mode

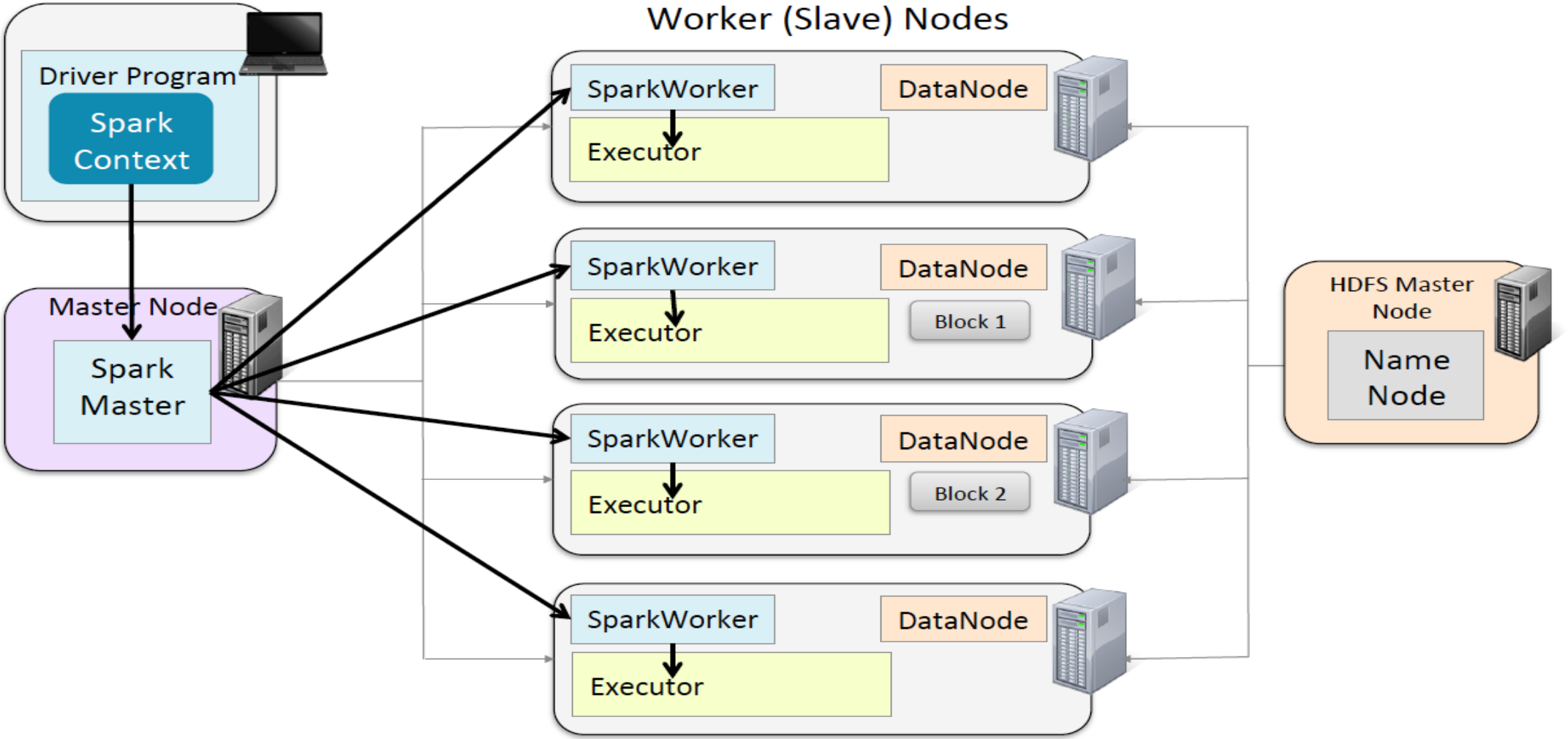


Standalone Mode

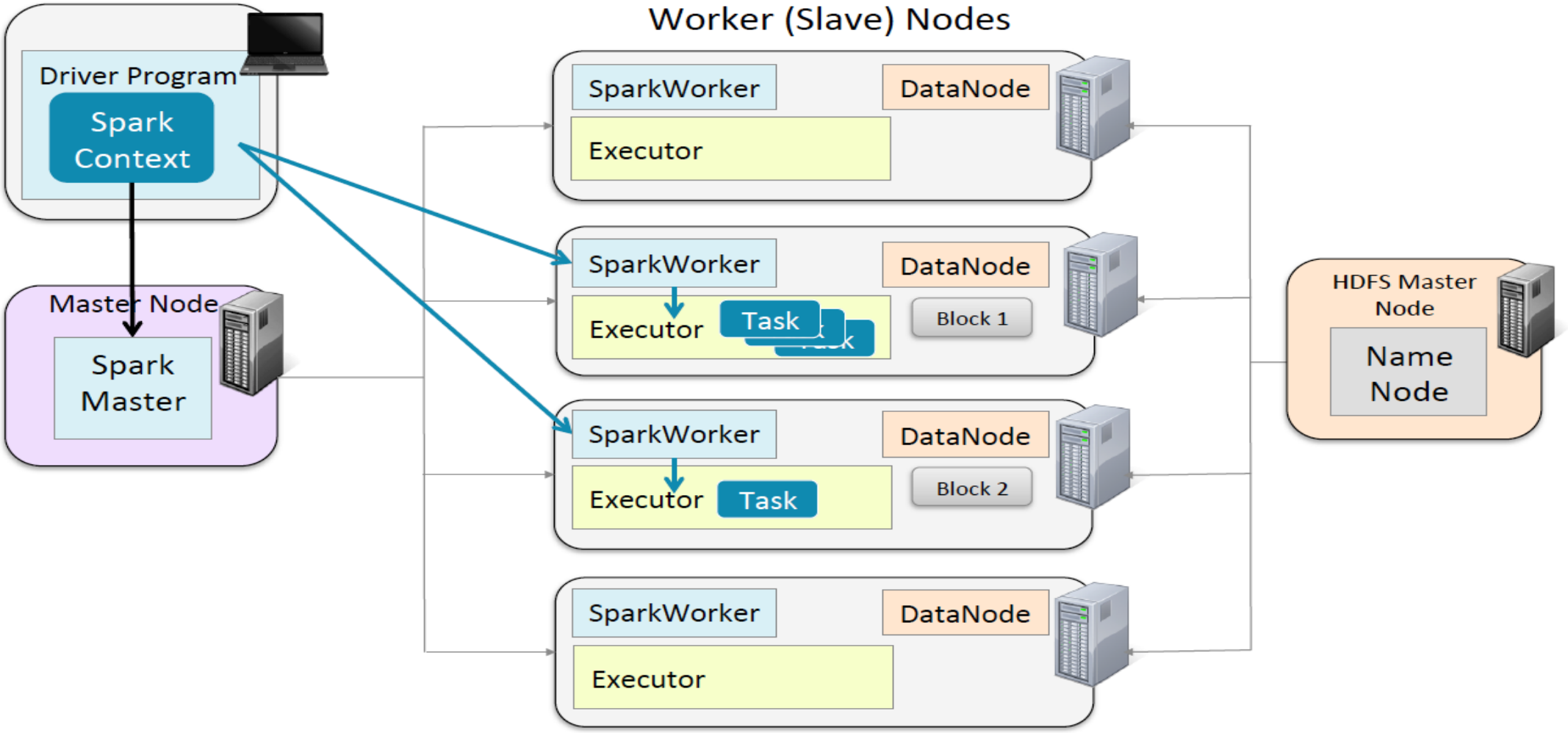
```
--executor-memory: RAM for each executor [Default 1GB]
--total-executor-cores: total CPU cores for this application [Default All cores]

spark-submit --class <classname with package name> <jarname>
--master spark://masternode:7077
```



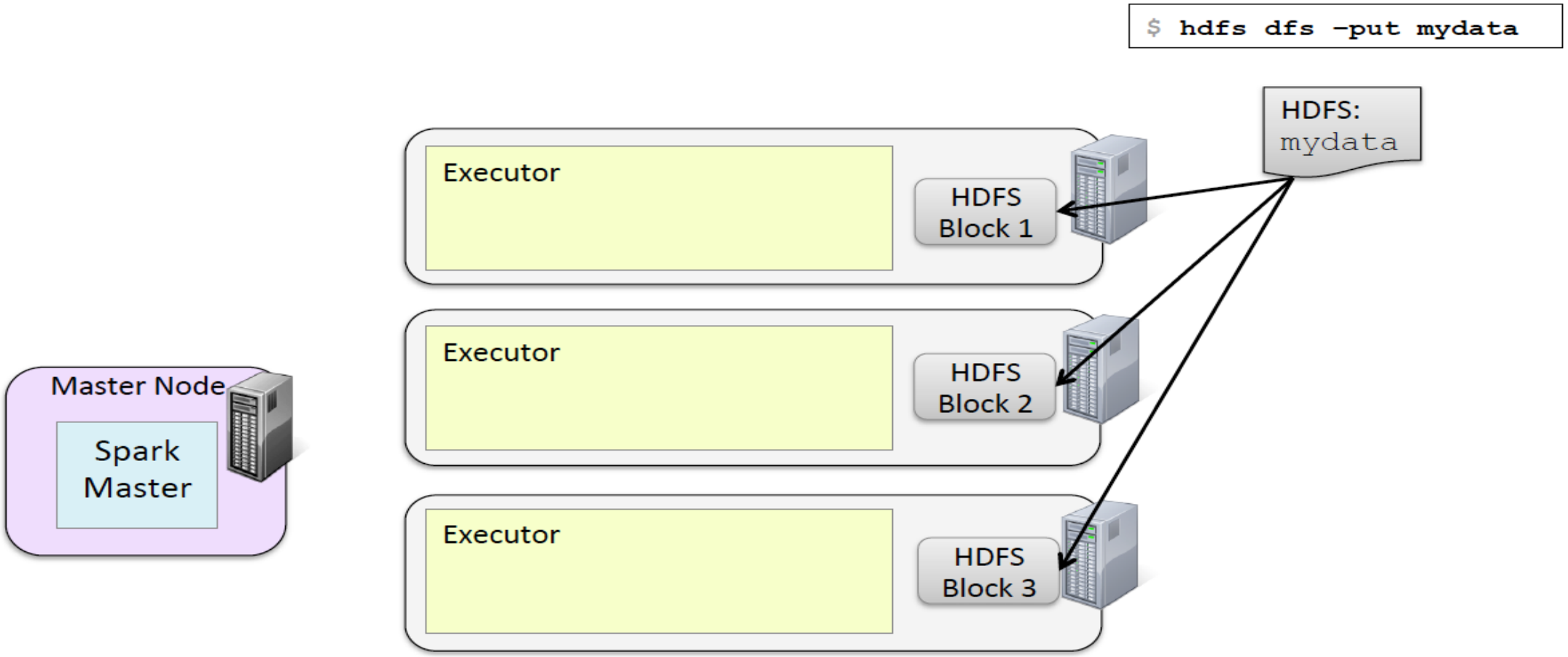
Running Spark on a Standalone Cluster (4)



Running Spark on a Standalone Cluster (5)



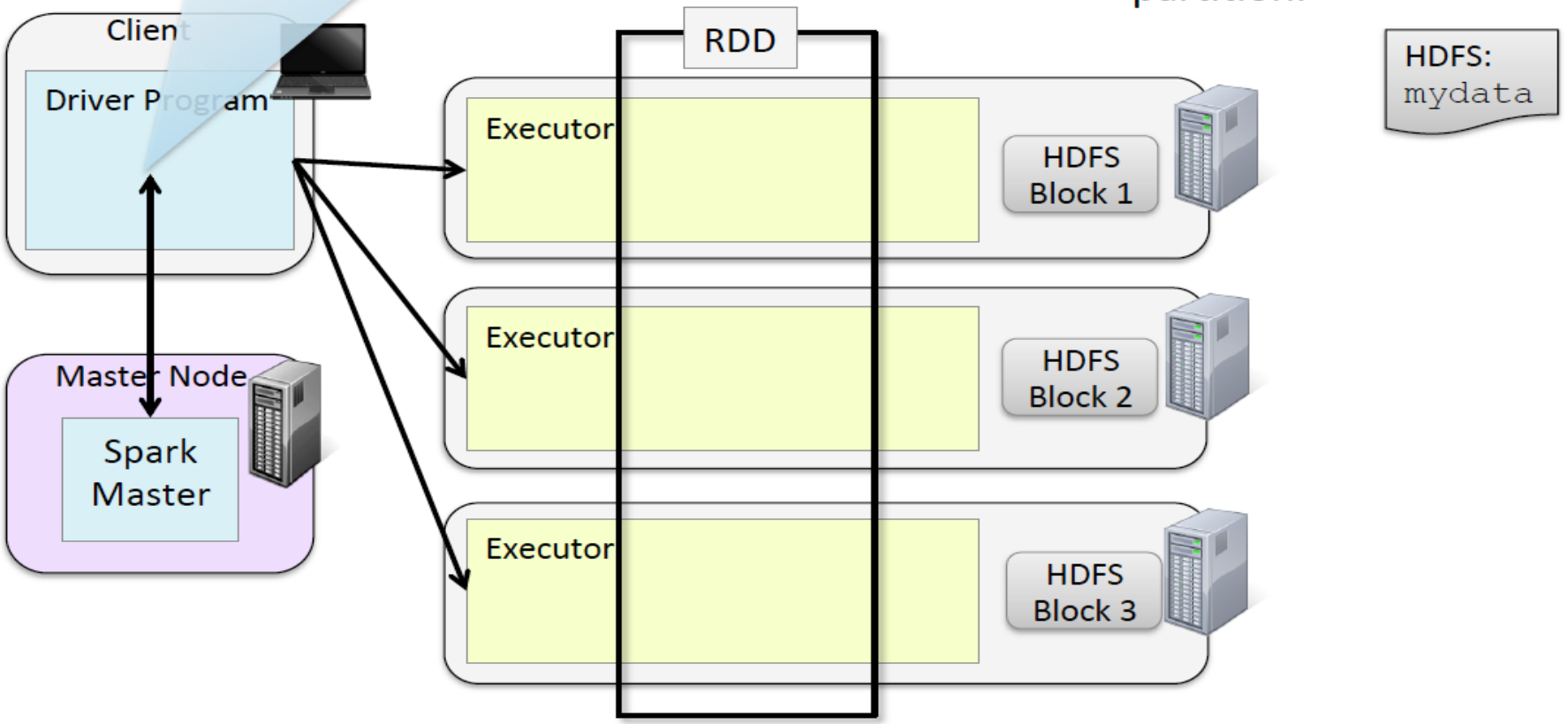
HDFS and Data Locality (1)



HDFS and Data Locality (2)

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

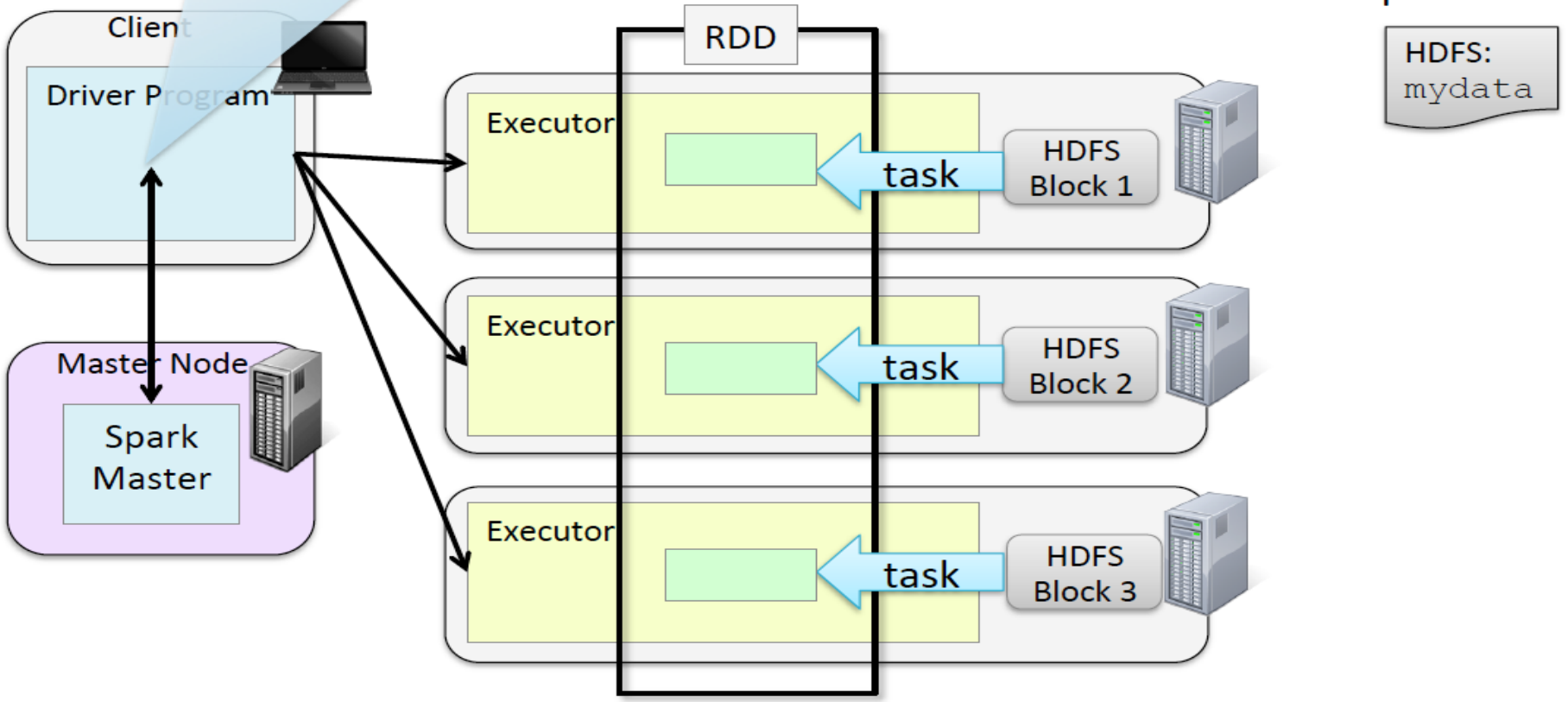
By default, Spark partitions file-based RDDs by block. Each block loads into a single partition.



HDFS and Data Locality (3)

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

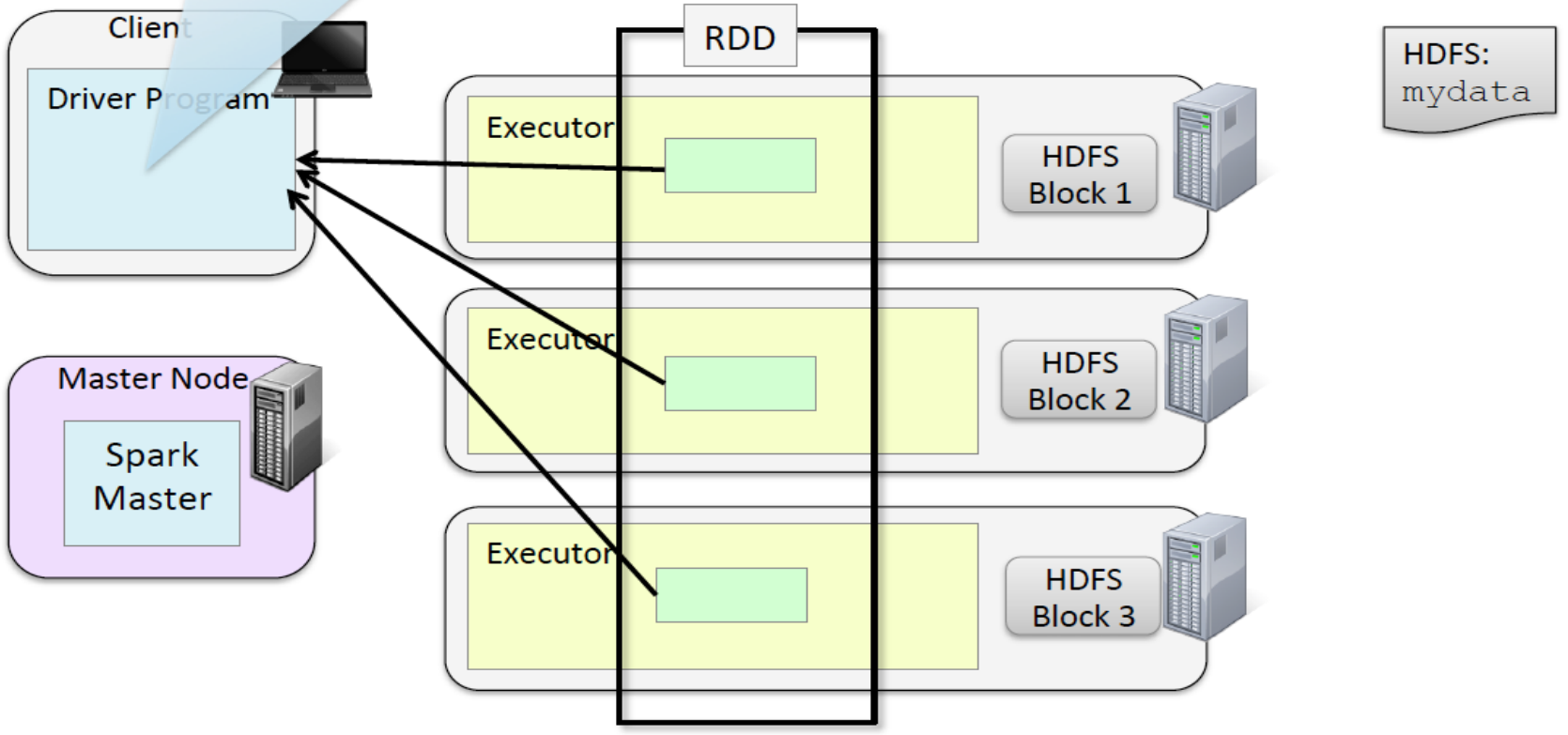
An action triggers execution: tasks on executors load data from blocks into partitions



HDFS and Data Locality (4)

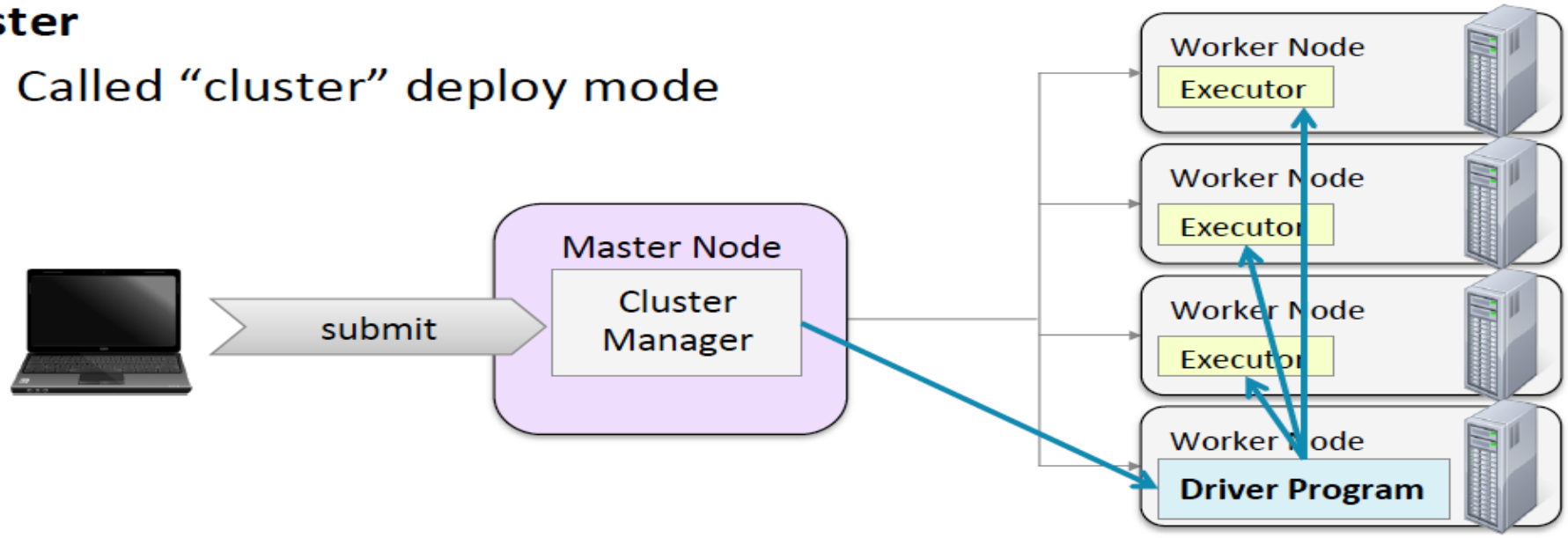
```
sc.textFile("hdfs://…mydata…").collect()
```

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



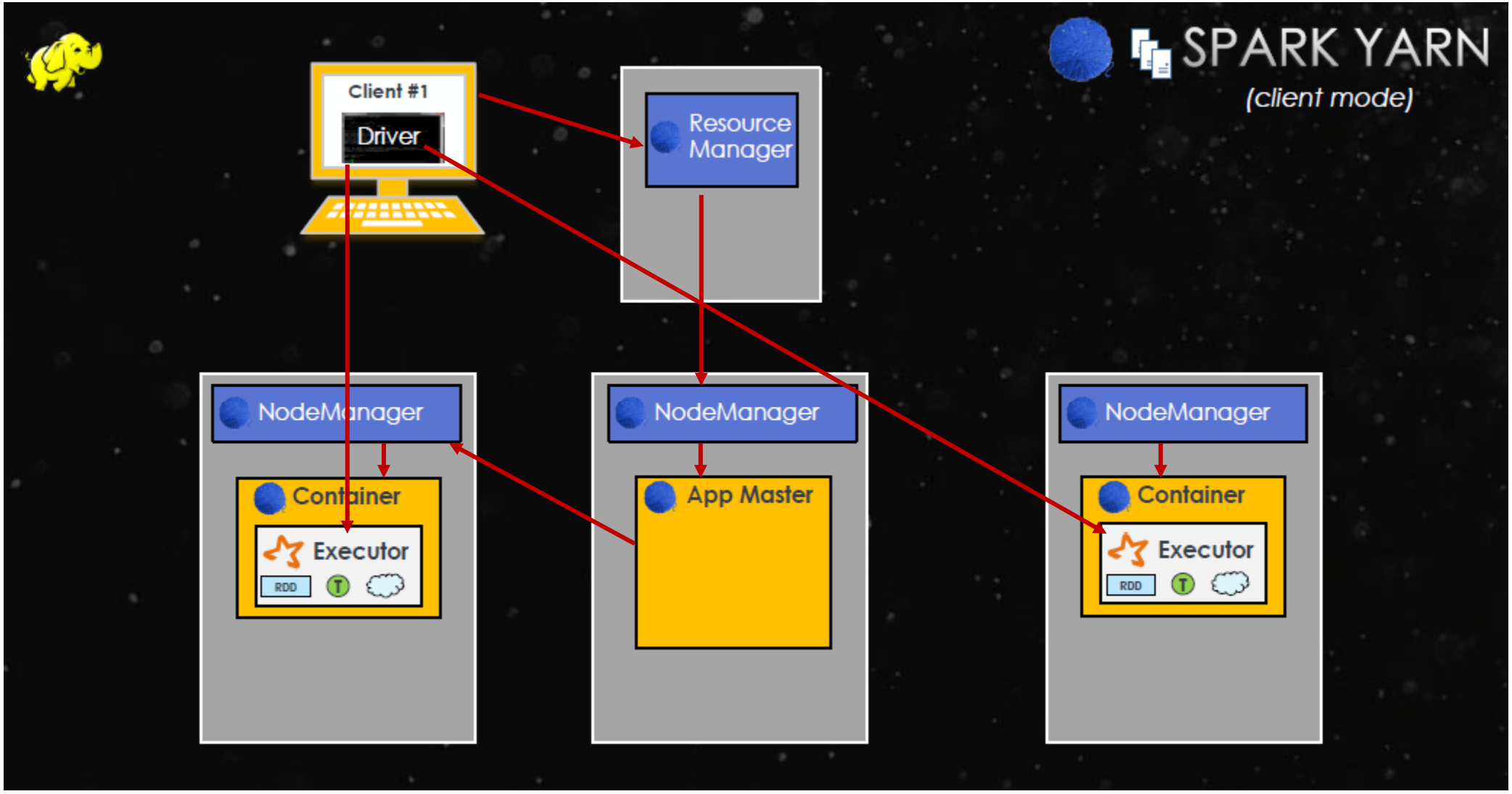
Client Mode and Cluster Mode

- **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

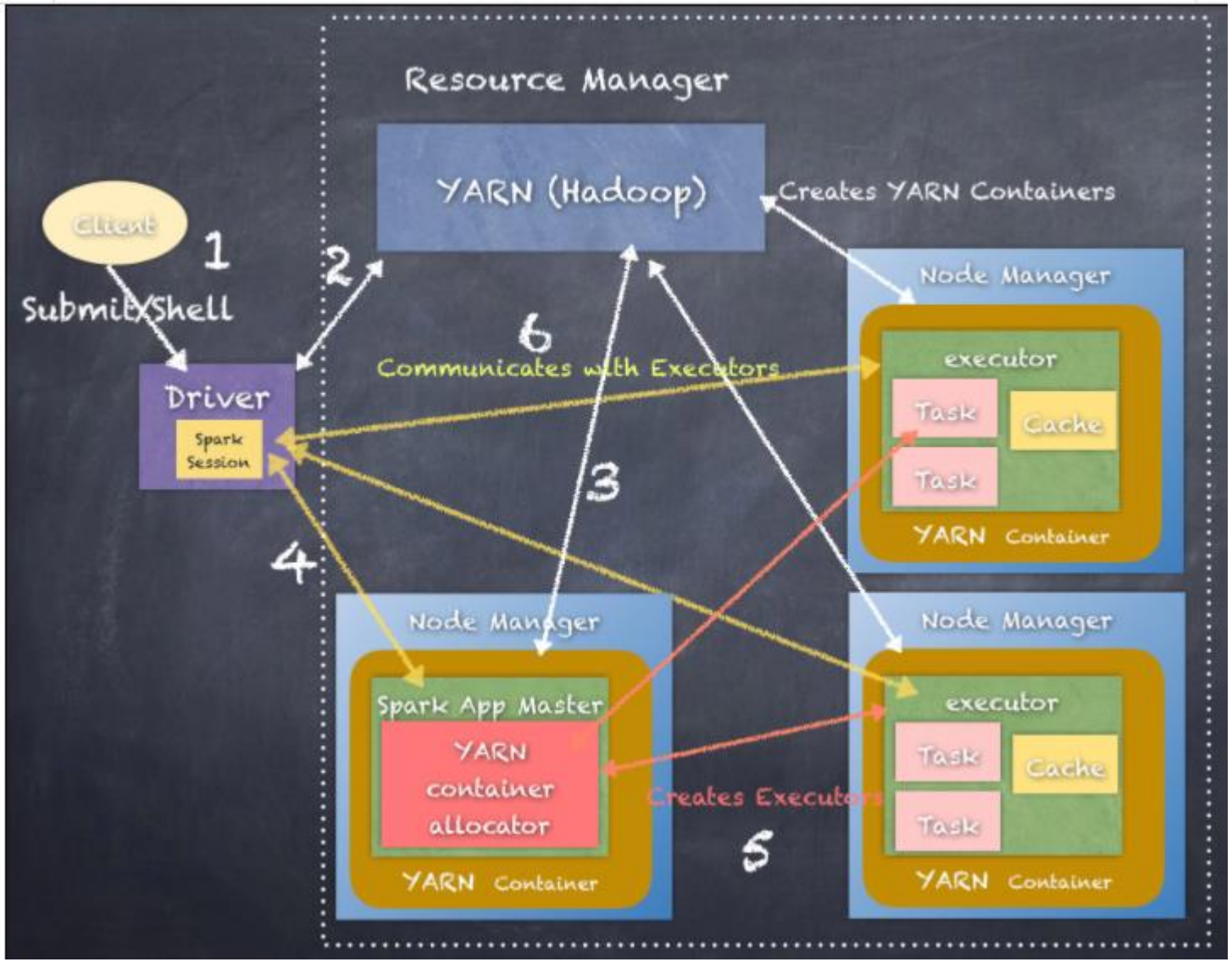


YARN – Client Mode

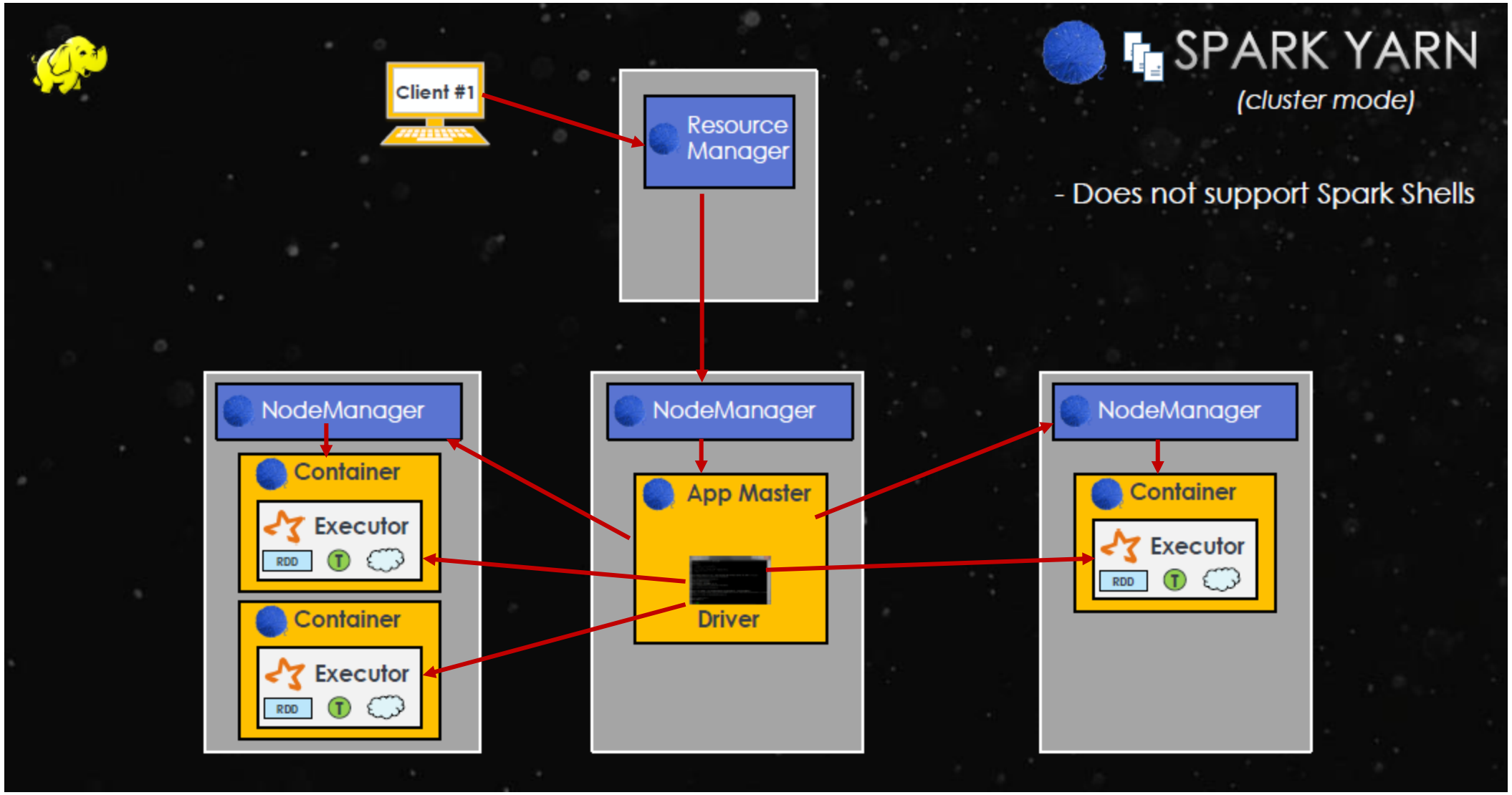
`--num-executors:` controls how many executors will be allocated
`--executor-memory:` RAM for each executor
`--executor-cores:` CPU cores for each executor



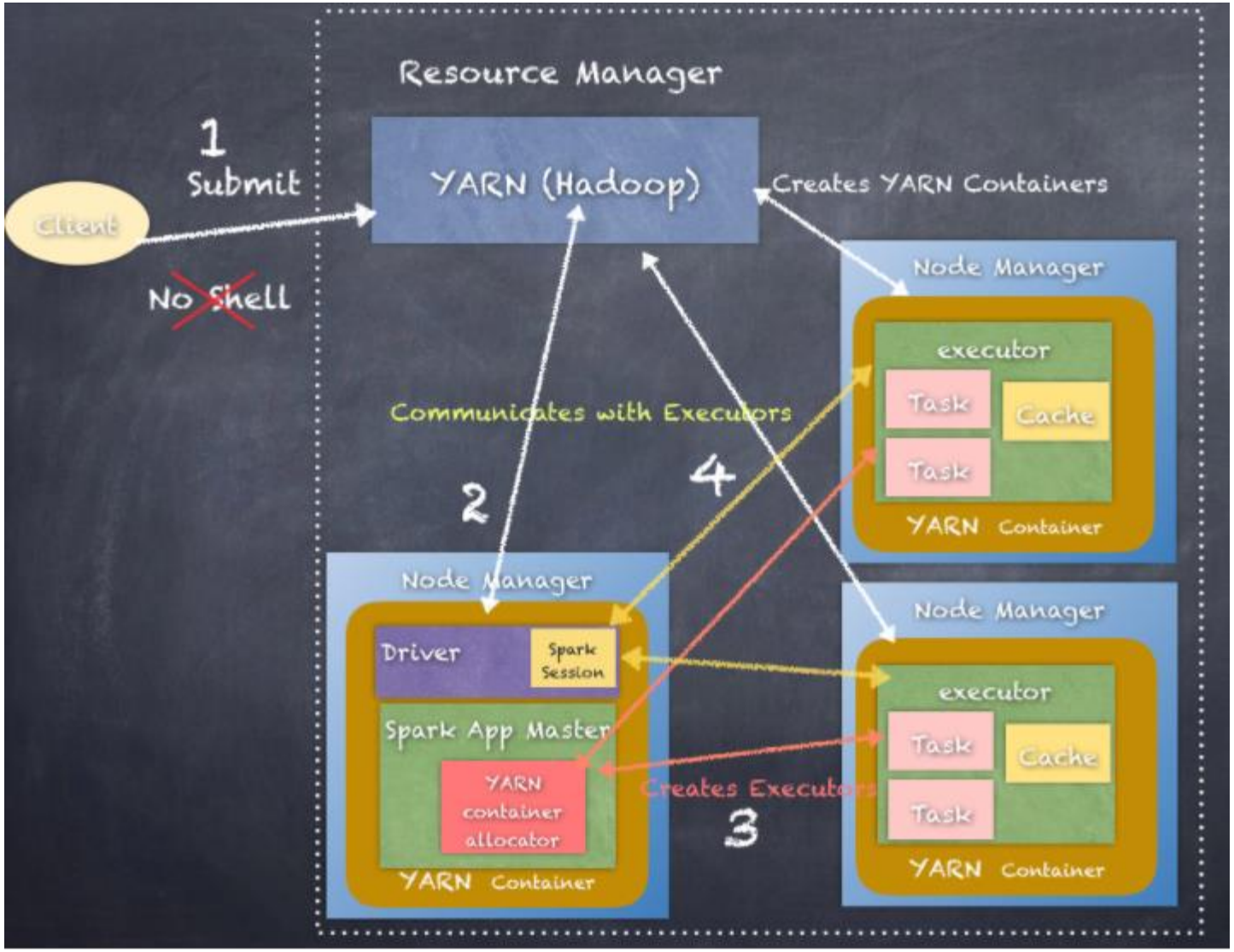
YARN – Client Mode – Application Life cycle



YARN – Cluster Mode



YARN – Cluster Mode – Application Life cycle



YARN – Static VS Dynamic Allocation

Static Allocation

- Traditional means of starting executors on nodes.

```
spark-shell --master yarn-client \
   --driver-memory 3686m \
   --executor-memory 17g \
   --executor-cores 7 \
   --num-executors 7
```

- Static number of executors specified by the submitter.
- Size and count of executors is key for good performance.

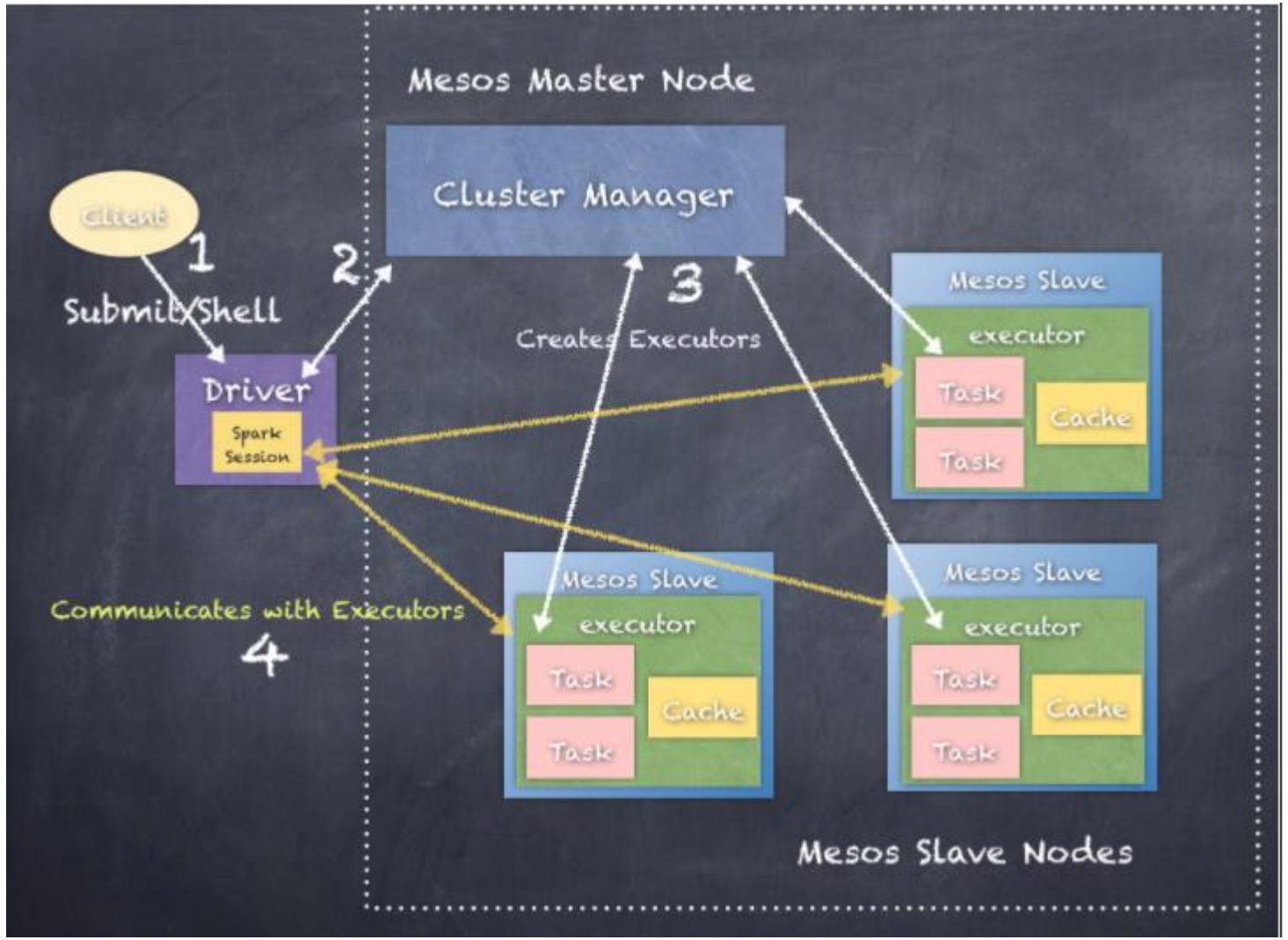
Dynamic Allocation

- Scale executor count based on pending tasks

```
spark-shell --master yarn-client \
   --driver-memory 3686m \
   --executor-memory 3686m \
   --executor-cores 1 \
   --conf "spark.dynamicAllocation.enabled=true" \
   --conf "spark.dynamicAllocation.minExecutors=1" \
   --conf "spark.dynamicAllocation.maxExecutors=100" \
   --conf "spark.shuffle.service.enabled=true"
```

- Minimum and maximum number of executors specified.
- Exclusive to running Spark on YARN

Mesos – Application Life cycle



Spark Configuration

spark-defaults.conf

Application Level Settings

Spark-env.sh

Per Machine Settings

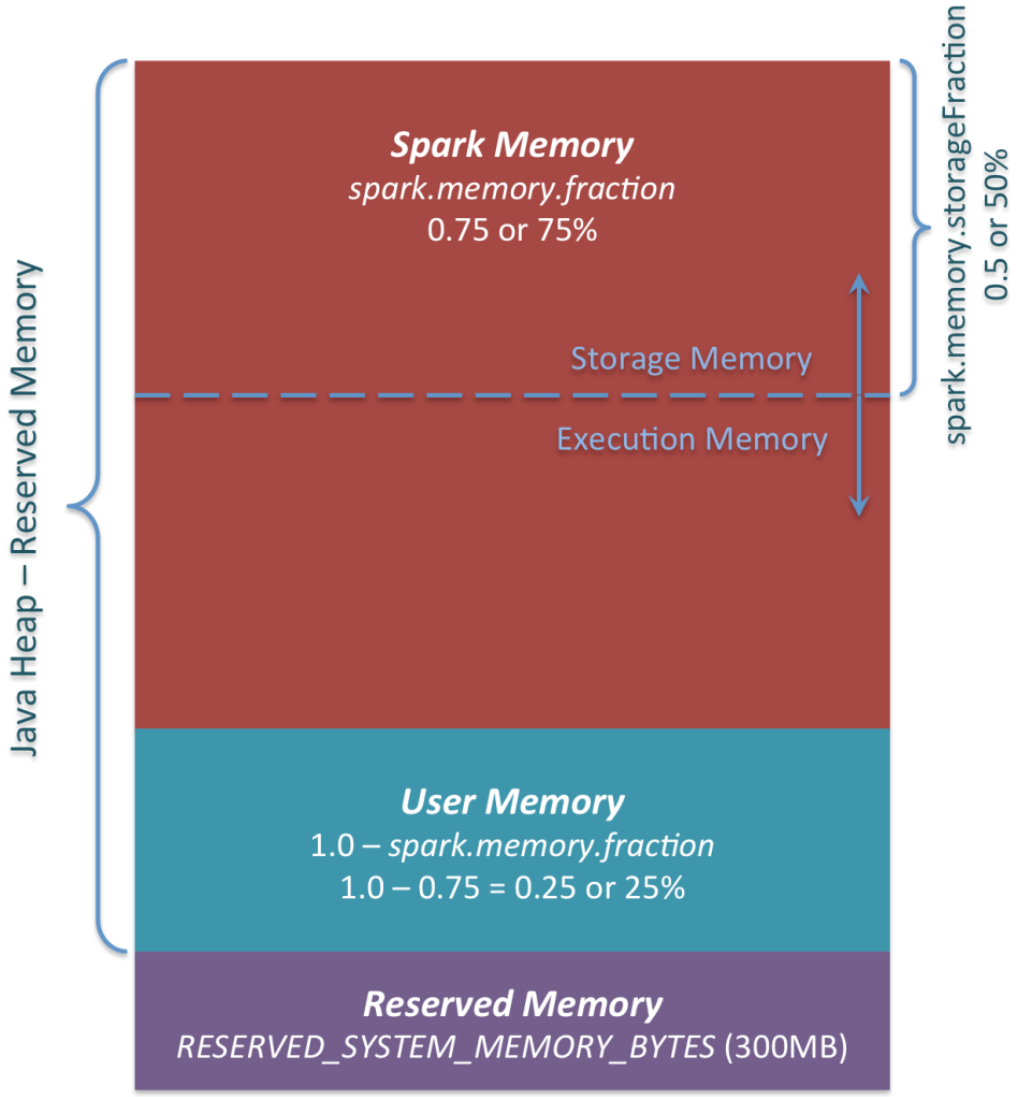
Log4j.properties

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**

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

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
```

CACHING AND PERSISTENCE

Persistence Levels

Transformation	Description
MEMORY_ONLY	RDD is stored as deserialized Java objects in the JVM. If the RDD does not fit into the memory, it will be only partially cached and missing partitions will be recomputed when they are required. This is the default mode.
MEMORY_AND_DISK	The RDD is stored as deserialized Java objects in the JVM. If the RDD does not fit into memory, it will be partially accessed from disk.
MEMORY_ONLY_SER	The RDD is stored as serialized Java objects (one by array for a partition). This approach is more byte-efficient than the former methods; however, requires more CPU to read. Recomputes missing partitions when required.
MEMORY_AND_DISK_SER	Similar to the above case, but uses disk instead of recomputing the missing partitions.
DISK_ONLY	Store the RDD only to disk.
MEMORY_ONLY_2...	Same as above, but with replication to two clusters.

Caching

- **Caching an RDD saves the data in memory**

File: purplecow.txt

```
I've never seen a purple cow.
I never hope to see one;
But I can tell you, anyhow,
I'd rather see than be one.
```

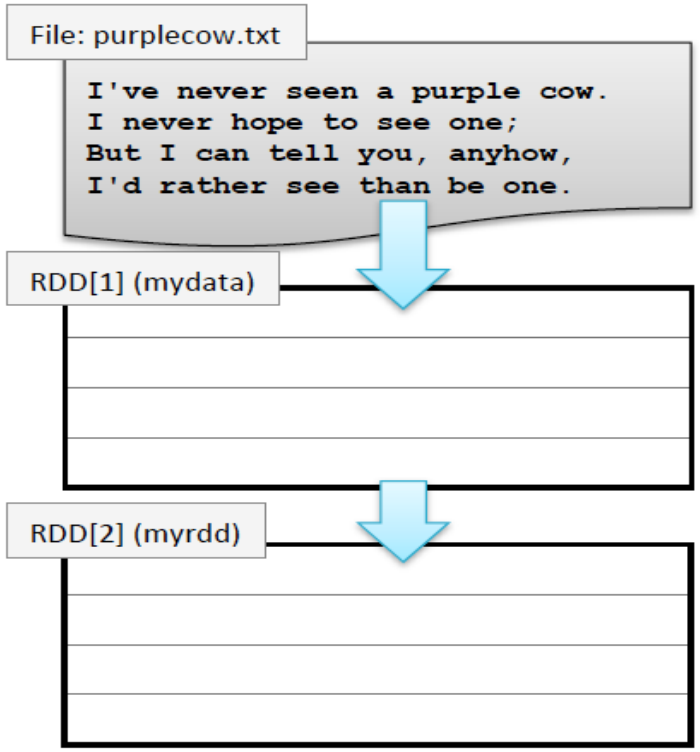
When and Where to Cache

- **When should you cache a dataset?**
  - When a dataset is likely to be re-used
    - e.g., iterative algorithms, machine learning
- **How to choose a persistence level**
  - Memory only – when possible, best performance
    - Save space by saving as serialized objects in memory if necessary
  - Disk – choose when recomputation is more expensive than disk read
    - e.g., expensive functions or filtering large datasets
  - Replication – choose when recomputation is more expensive than memory

Caching

- **Caching an RDD saves the data in memory**

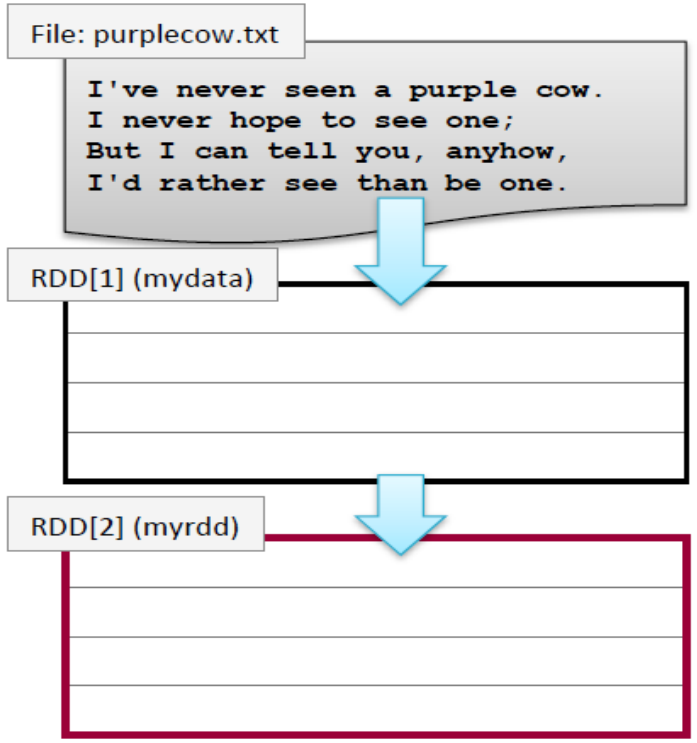
```
> mydata = sc.textFile("purplecow.txt")
> myrdd = mydata.map(lambda s:
  s.upper())
```



Caching

- **Caching an RDD saves the data in memory**

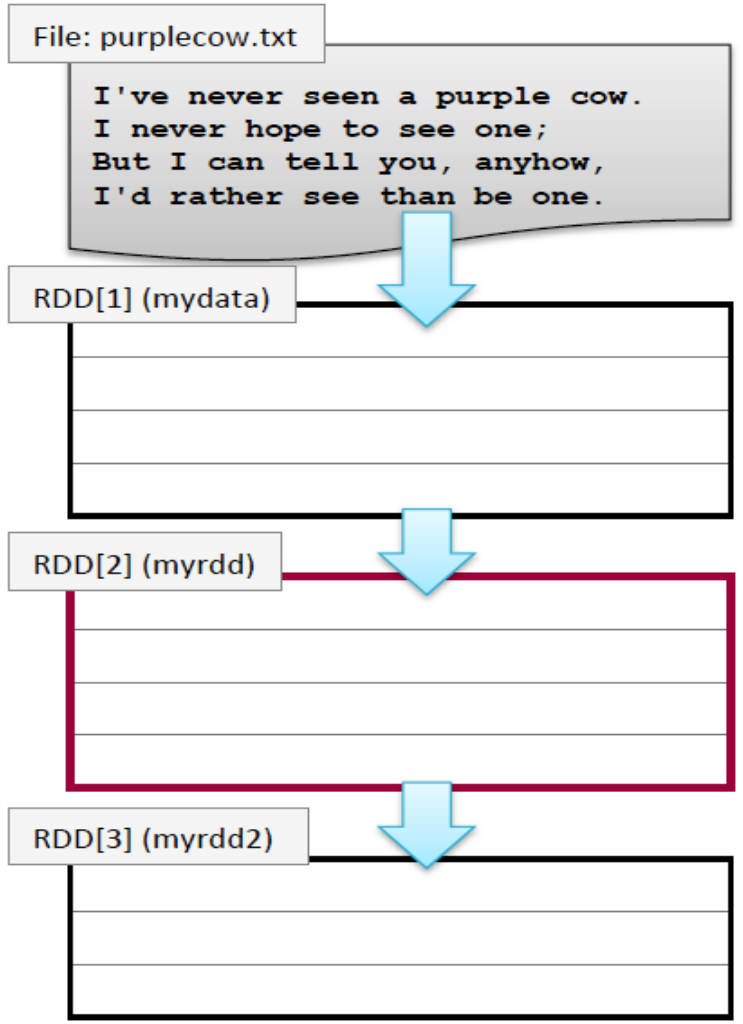
```
> mydata = sc.textFile("purplecow.txt")
> myrdd = mydata.map(lambda s:
  s.upper())
> myrdd.cache()
```



Caching

- **Caching an RDD saves the data in memory**

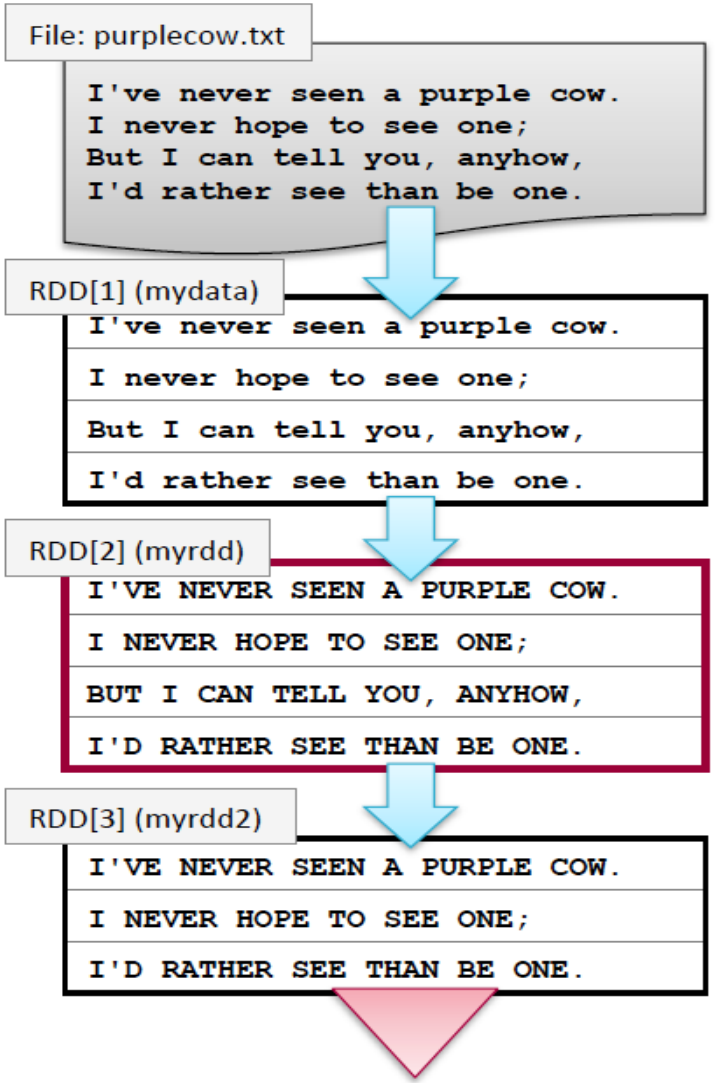
```
> mydata = sc.textFile("purplecow.txt")
> myrdd = mydata.map(lambda s:
  s.upper())
> myrdd.cache()
> myrdd2 = myrdd.filter(lambda \
  s:s.startswith('I'))
```



Caching

- **Caching an RDD saves the data in memory**

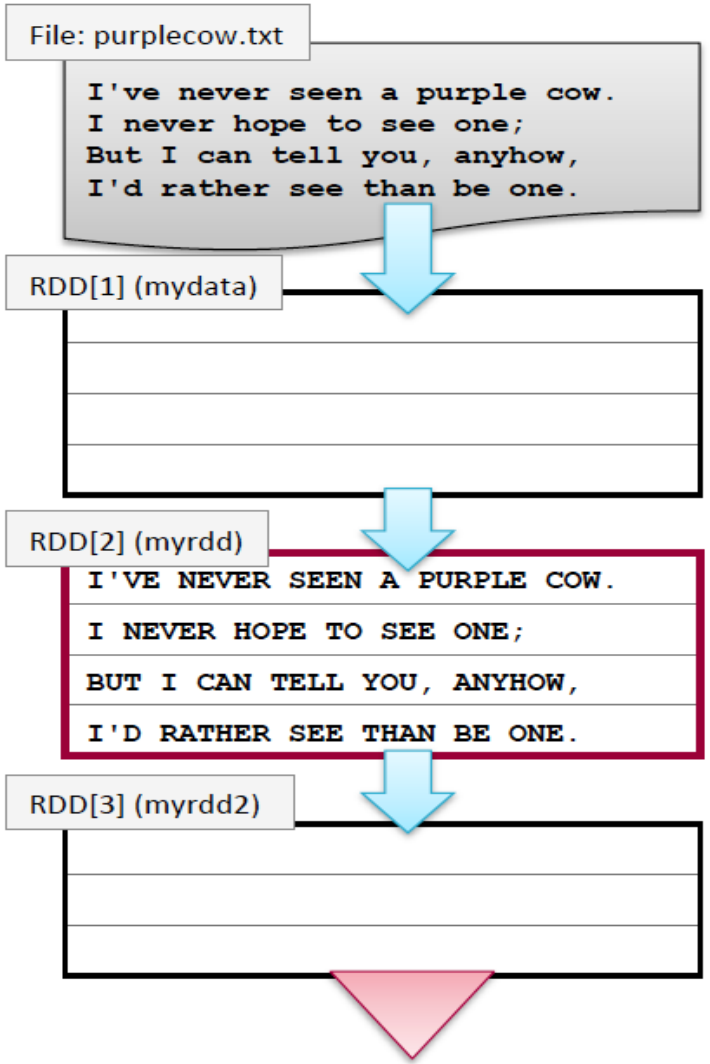
```
> mydata = sc.textFile("purplecow.txt")
> myrdd = mydata.map(lambda s:
  s.upper())
> myrdd.cache()
> myrdd2 = myrdd.filter(lambda \
  s:s.startswith('I'))
> myrdd2.count()
3
```



Caching

- **Subsequent operations use saved data**

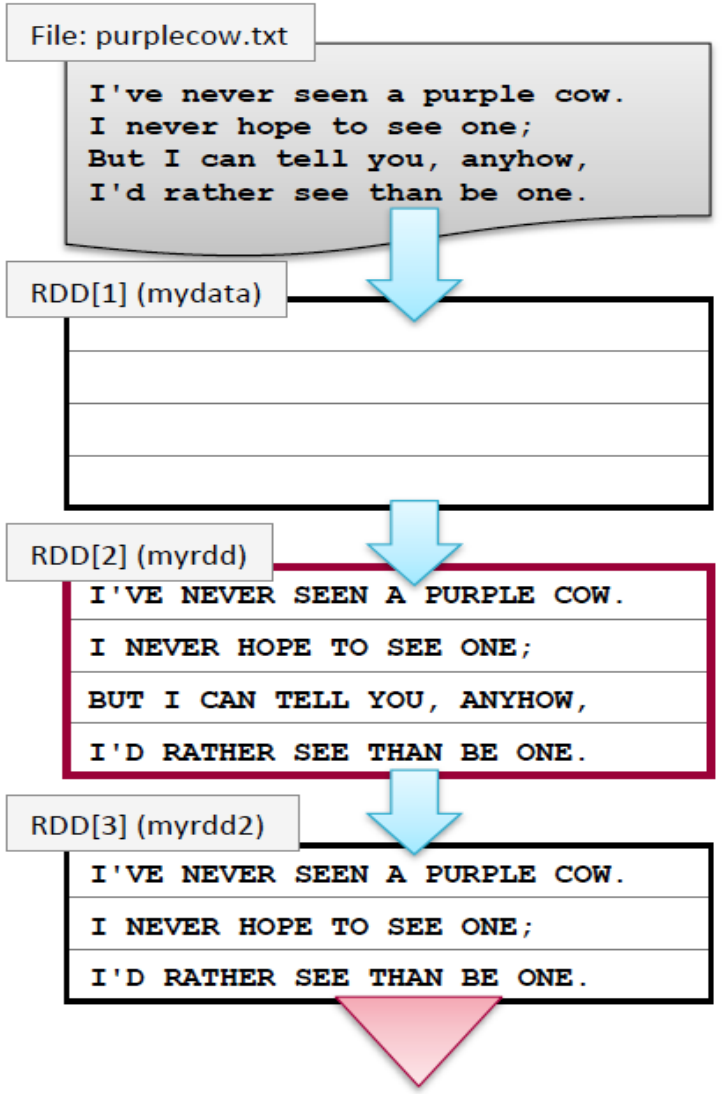
```
> mydata = sc.textFile("purplecow.txt")
> myrdd = mydata.map(lambda s:
  s.upper())
> myrdd.cache()
> myrdd2 = myrdd.filter(lambda \
  s:s.startswith('I'))
> myrdd2.count()
3
> myrdd2.count()
```



Caching

- **Subsequent operations use saved data**

```
> mydata = sc.textFile("purplecow.txt")
> myrdd = mydata.map(lambda s:
  s.upper())
> myrdd.cache()
> myrdd2 = myrdd.filter(lambda \
  s:s.startswith('I'))
> myrdd2.count()
3
> myrdd2.count()
3
```

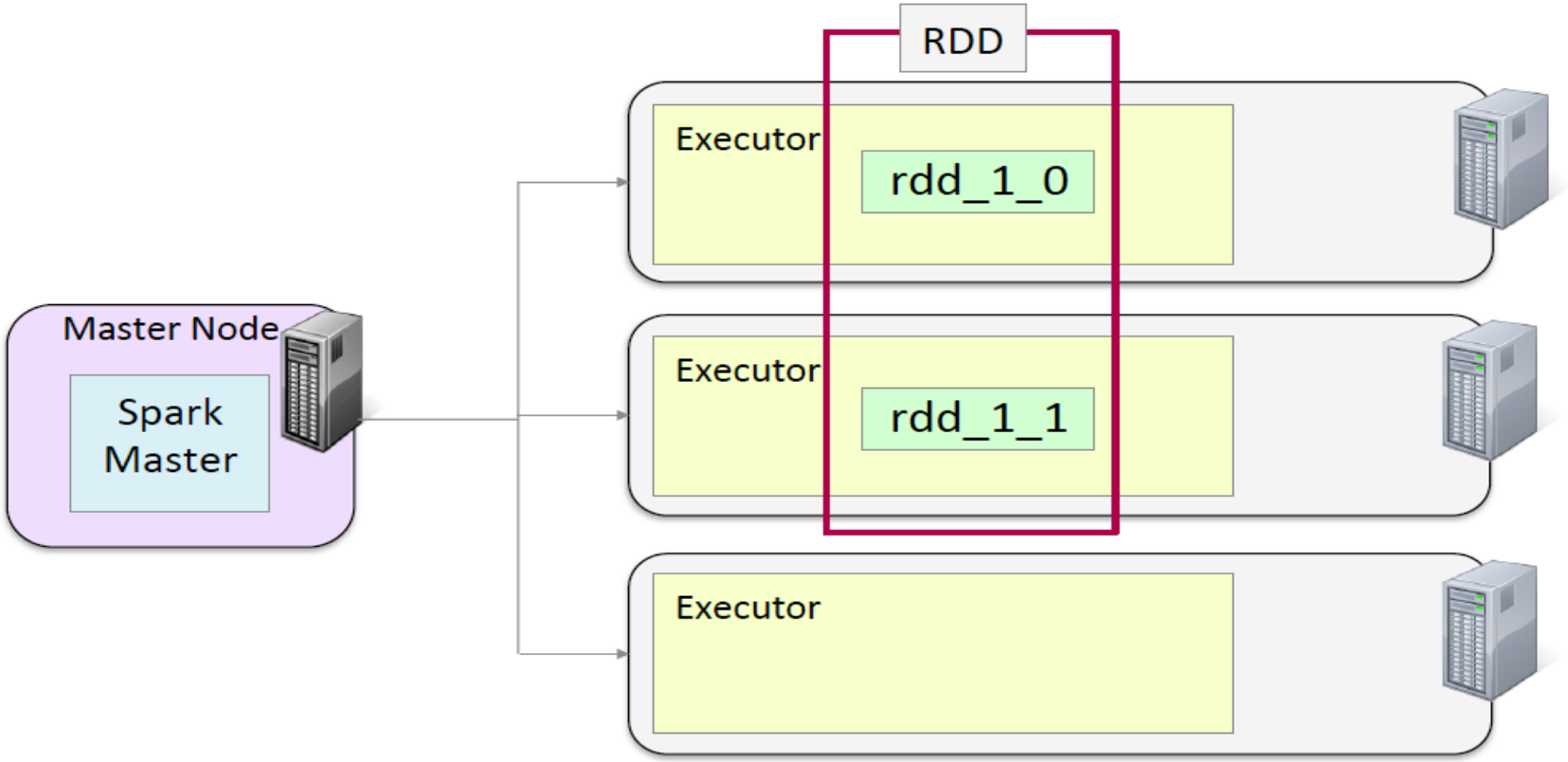


Caching and Fault-Tolerance

- **RDD = *Resilient* Distributed Dataset**
  - Resiliency is a product of tracking lineage
  - RDDs can always be recomputed from their base if needed

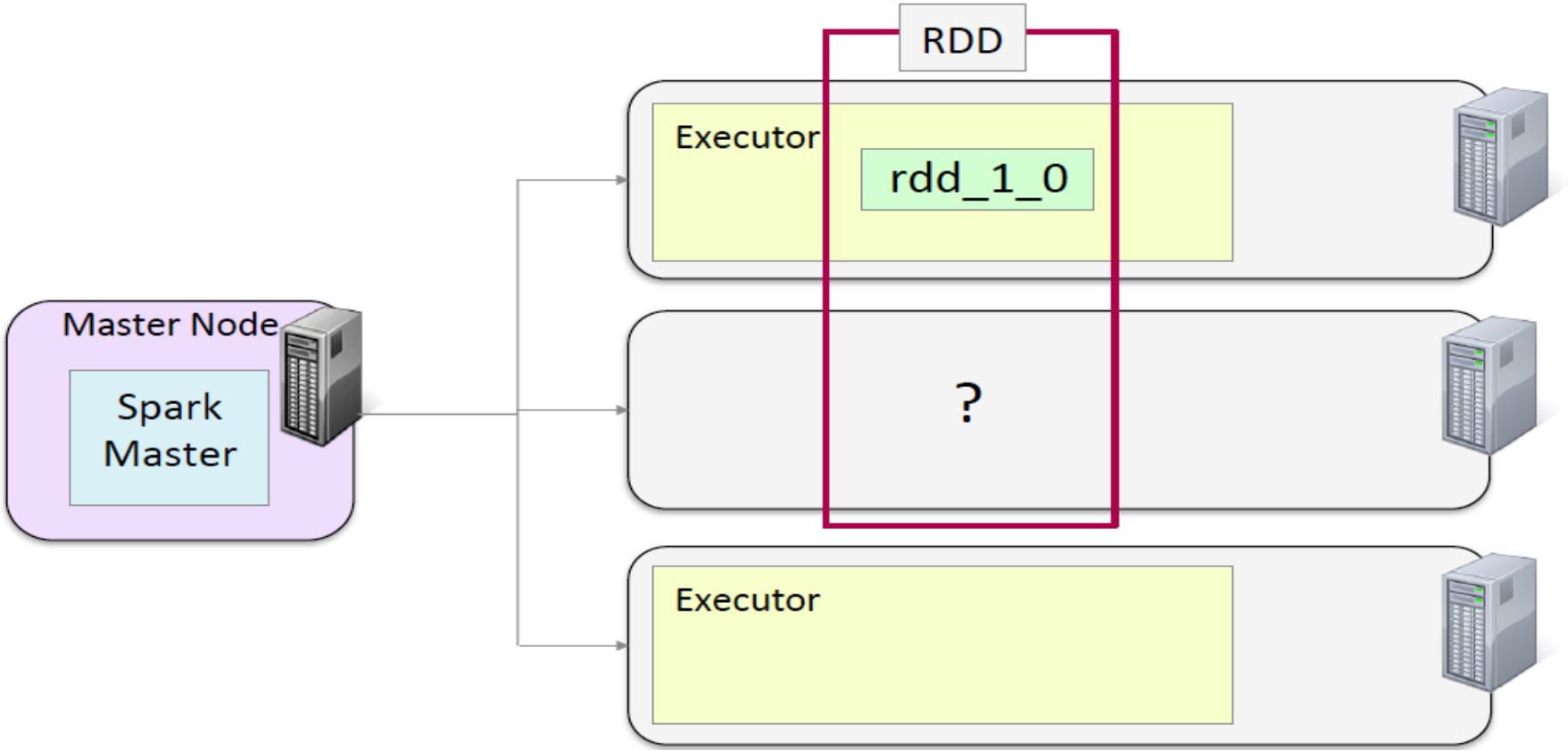
Distributed Cache

- **RDD partitions are distributed across a cluster**
- **Cached partitions are stored in memory in Executor JVMs**



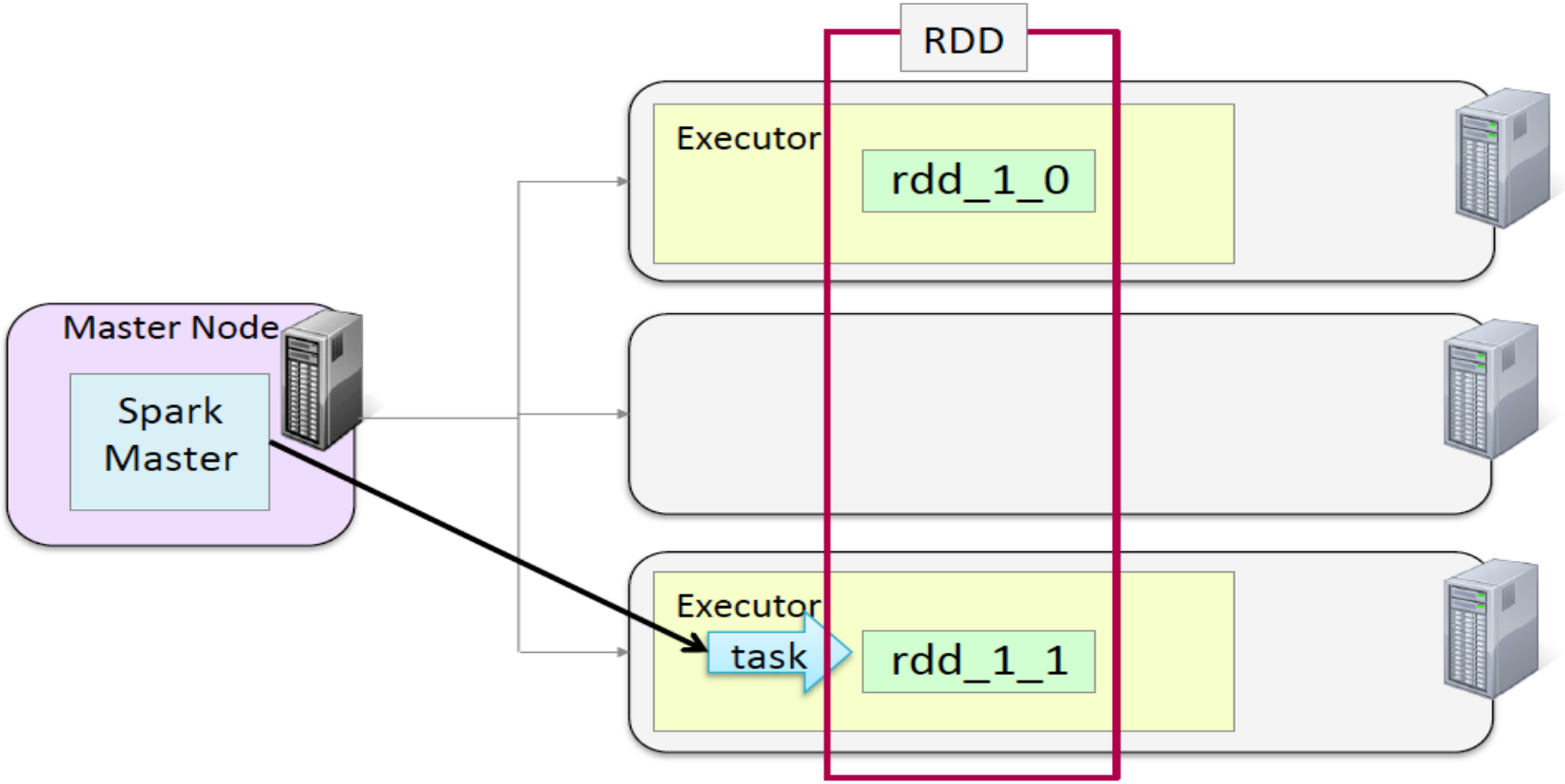
RDD Fault-Tolerance (1)

- **What happens if a cached partition becomes unavailable?**



RDD Fault-Tolerance (2)

- **The SparkMaster starts a new task to recompute the partition on a different node**



Persistence Levels (1)

- **The `cache` method stores data in memory only**
- **The `persist` method offers other options called Storage Levels**
- **Storage location – where is the data stored?**
  - **`MEMORY_ONLY`** (default) – same as **`cache`**
  - **`MEMORY_AND_DISK`** – Store partitions on disk if they do not fit in memory
    - Called *spilling*
  - **`DISK_ONLY`** – Store all partitions on disk
- **Replication – store partitions on two nodes**
  - **`MEMORY_ONLY_2`**, **`MEMORY_AND_DISK_2`**, etc.

Persistence Levels (2)

- **Serialization – you can choose to serialize the data in memory**
  - **`MEMORY_ONLY_SER`** and **`MEMORY_AND_DISK_SER`**
  - Much more space efficient
  - Less time efficient
    - Choose a fast serialization library (covered later)

Python

```
> from pyspark import StorageLevel
> myrdd.persist(StorageLevel.DISK_ONLY)
```

Scala

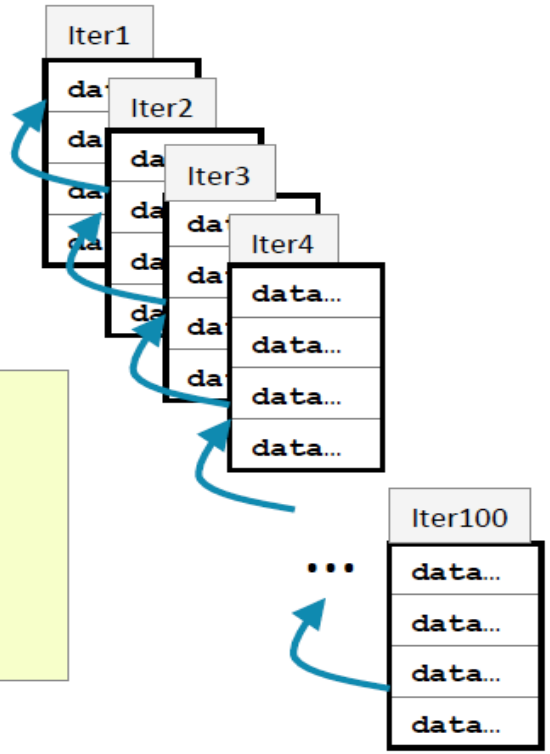
```
> import org.apache.spark.storage.StorageLevel
> myrdd.persist(StorageLevel.DISK_ONLY)
```

Changing Persistence Options

- **To stop persisting and remove from memory and disk**
  - `rdd.unpersist()`
- **To change an RDD to a different persistence level**
  - Unpersist first

Checkpointing (1)

- **Maintaining RDD lineage provides resilience but can also cause problems when the lineage gets very long**
  - e.g., iterative algorithms, streaming
- **Recovery can be very expensive**
- **Potential stack overflow**

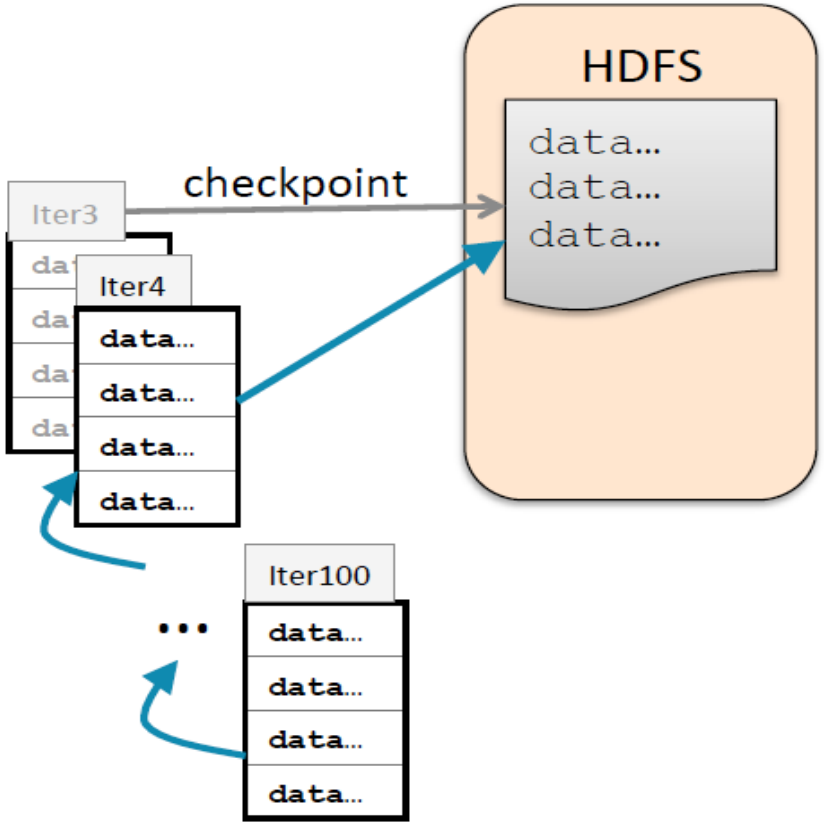



```
myrdd = ...initial-value...
while x in xrange(100):
    myrdd = myrdd.transform(...)
myrdd.saveAsTextFile()
```

Checkpointing (2)

- **Checkpointing saves the data to HDFS**
  - Provides fault-tolerant storage across nodes
- **Lineage is not saved**
- **Must be checkpointed before any actions on the RDD**

```
sc.setCheckpointDir(directory)
myrdd = ...initial-value...
while x in xrange(100):
  myrdd = myrdd.transform(...)
  if x % 3 == 0:
    myrdd.checkpoint()
    myrdd.count()
myrdd.saveAsTextFile()
```

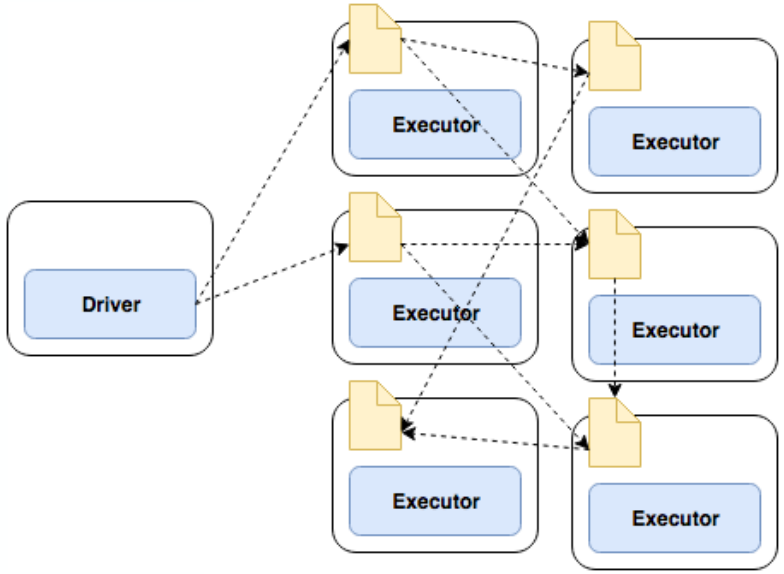


Spark Essentials : Broadcast Variables

Broadcast variables let programmer keep a read-only variable cached on each machine rather than shipping a copy of it with tasks

For example, to give every node a copy of a large input dataset efficiently

Spark also attempts to distribute broadcast variables using efficient broadcast algorithms to reduce communication cost



In Spark broadcast variables uses Torrent protocol.

Torrent protocol is a Peer-to-Peer protocol perform very well for distributing data sets across multiple peers.

INCEPTEZ TECHNOLOGIES

Spark Essentials : Broadcast Variables

```
case class Transaction(id: Long, custId: Int, itemId: Int)
case class TransactionDetail(id: Long, custName: String, itemName: String)

val customerMap = Map(1 -> "Tom", 2 -> "Harry")
val itemMap = Map(1 -> "Razor", 2 -> "Blade")
val transactions = sc.parallelize(List(Transaction(1, 1, 1), Transaction(2, 1, 2)))

val bcCustomerMap = sc.broadcast(customerMap)
val bcItemMap = sc.broadcast(itemMap)

val transactionDetails = transactions.map{t => TransactionDetail(
t.id, bcCustomerMap.value(t.custId), bcItemMap.value(t.itemId))}

transactionDetails.collect
```

Spark Essentials : Accumulators

Accumulators are variables that can only be "added" to through an *associative* operation

Used to implement counters and sums, efficiently in parallel

Spark natively supports accumulators of numeric value types and standard mutable collections, and programmers can extend for new types

Only the driver program can read an accumulator's value, not the tasks

INCEPTEZ TECHNOLOGIES

Spark Essentials : Accumulators

Scala:

```scala
val accum = sc.accumulator(0)
sc.parallelize(Array(1, 2, 3, 4)).foreach(x => accum += x)

accum.value
```

Python:

```python
accum = sc.accumulator(0)
rdd = sc.parallelize([1, 2, 3, 4])
def f(x):
    global accum
    accum += x

rdd.foreach(f)

accum.value
```

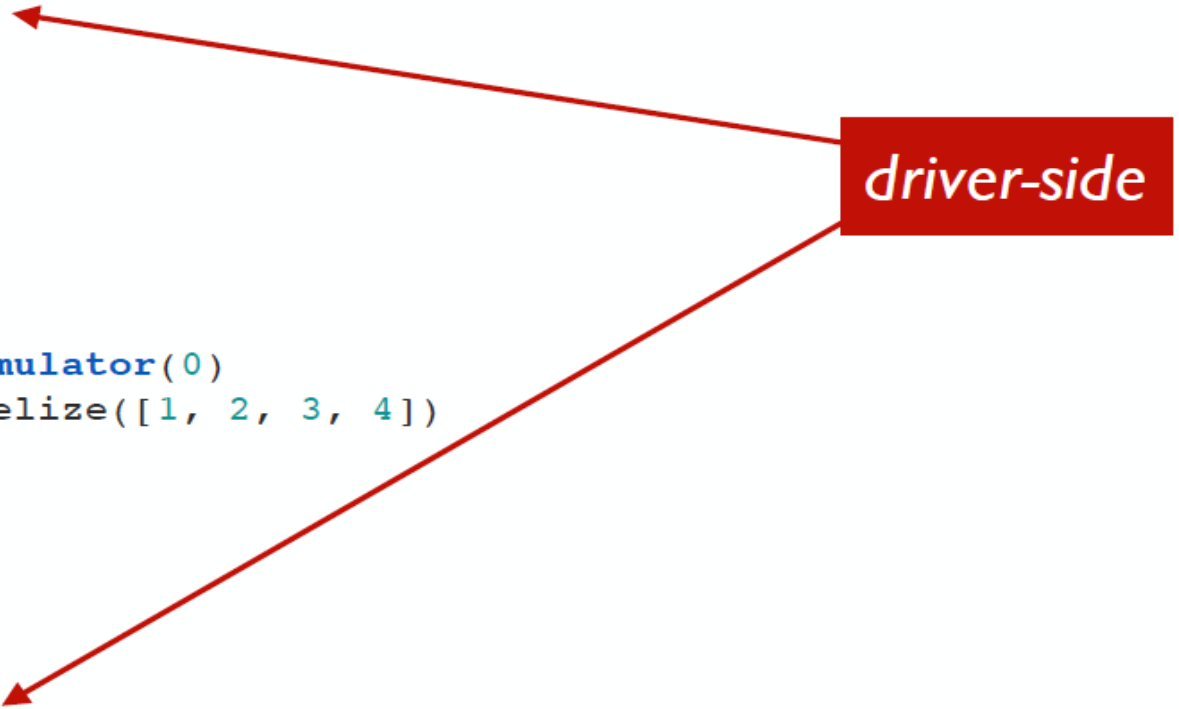
Spark Essentials : Accumulators

Scala:

```scala
val accum = sc.accumulator(0)
sc.parallelize(Array(1, 2, 3, 4)).foreach(x => accum += x)

accum.value
```

Python:

```python
accum = sc.accumulator(0)
rdd = sc.parallelize([1, 2, 3, 4])
def f(x):
    global accum
    accum += x

rdd.foreach(f)

accum.value
```



Serialization

Choice of Serializer

Serialization is sometimes a bottleneck when shuffling and caching data. Using the Kryo serializer is often faster.

```
val conf = new SparkConf()
conf.set("spark.serializer", "org.apache.spark.serializer.KryoSerializer")

// Be strict about class registration
conf.set("spark.kryo.registrationRequired", "true")
conf.registerKryoClasses(Array(classOf[MyClass],
classOf[MyOtherClass]))
```

The official Spark Documentation says this:

> The only reason Kryo is not the default is because of the custom registration requirement, but we recommend trying it in any network-intensive application.
>
> Since Spark 2.0.0, we internally use Kryo serializer when shuffling RDDs with simple types, arrays of simple types, or string type

Performance Debugging

Distributed performance: program slow due to scheduling, coordination, or data distribution)

Local performance: program slow because whatever I'm running is just slow on a single node

Two useful tools:

- Application web UI (default port 4040)
- Executor logs (spark/work)

Stragglers due to slow nodes

Stages for All Jobs

Completed Stages: 18

Completed Stages (18)

Stage Id	Description		Submitted	Duration	Tasks: Succeeded/Total	Input	Output	Shuffle Read	Shuffle Write
19	collect at FireServiceCallAnalysisDF.scala:83	+details	2017/06/17 09:07:48	0.1 s	23/23			3.3 KB	
18	collect at FireServiceCallAnalysisDF.scala:83	+details	2017/06/17 09:07:48	0.1 s	23/23			7.1 KB	3.3 KB
16	collect at FireServiceCallAnalysisDF.scala:83	+details	2017/06/17 09:07:48	0.1 s	23/23			7.1 KB	
15	collect at FireServiceCallAnalysisDF.scala:83	+details	2017/06/17 09:07:48	0.6 s	23/23	849.2 MB			7.1 KB
14	show at FireServiceCallAnalysisDF.scala:78	+details	2017/06/17 09:07:47	0.3 s	23/23			109.4 KB	
13	show at FireServiceCallAnalysisDF.scala:78	+details	2017/06/17 09:07:46	0.7 s	23/23	849.2 MB			109.4 KB
12	show at FireServiceCallAnalysisDF.scala:73	+details	2017/06/17 09:07:46	0.3 s	23/23			2.6 KB	
11	show at FireServiceCallAnalysisDF.scala:73	+details	2017/06/17 09:07:45	0.6 s	23/23	849.2 MB			2.6 KB
10	collect at FireServiceCallAnalysisDF.scala:68	+details	2017/06/17 09:07:45	0.2 s	23/23			2.7 KB	
9	collect at FireServiceCallAnalysisDF.scala:68	+details	2017/06/17 09:07:44	0.3 s	23/23			45.7 KB	2.7 KB
7	collect at FireServiceCallAnalysisDF.scala:68	+details	2017/06/17 09:07:44	0.3 s	23/23			45.7 KB	
6	collect at FireServiceCallAnalysisDF.scala:68	+details	2017/06/17 09:07:43	1 s	23/23	849.2 MB			45.7 KB
5	collect at FireServiceCallAnalysisDF.scala:63	+details	2017/06/17 09:07:42	0.3 s	23/23			40.7 KB	
4	collect at FireServiceCallAnalysisDF.scala:63	+details	2017/06/17 09:07:41	1.0 s	23/23	849.2 MB			40.7 KB
3	count at FireServiceCallAnalysisDF.scala:57	+details	2017/06/17 09:07:41	0.1 s	1/1			1357.0 B	
2	count at FireServiceCallAnalysisDF.scala:57	+details	2017/06/17 09:06:55	46 s	23/23	1352.5 MB			1357.0 B
1	show at FireServiceCallAnalysisDF.scala:54	+details	2017/06/17 09:06:55	46 ms	1/1	38.8 MB			
0	take at FireServiceCallAnalysisDF.scala:48	+details	2017/06/17 09:06:45	10 s	1/1				

Stragglers due to slow nodes

Turn speculation on to mitigates this problem.

Speculation: Spark identifies slow tasks (by looking at runtime distribution), and re-launches those tasks on other nodes.

spark.speculation true

HANDS ON