Introduction to Distributed Systems

Morning Part 1
Goals

- Local vs Distributed
- Distributed Systems Architecture
- Hadoop File System & Hadoop MapReduce

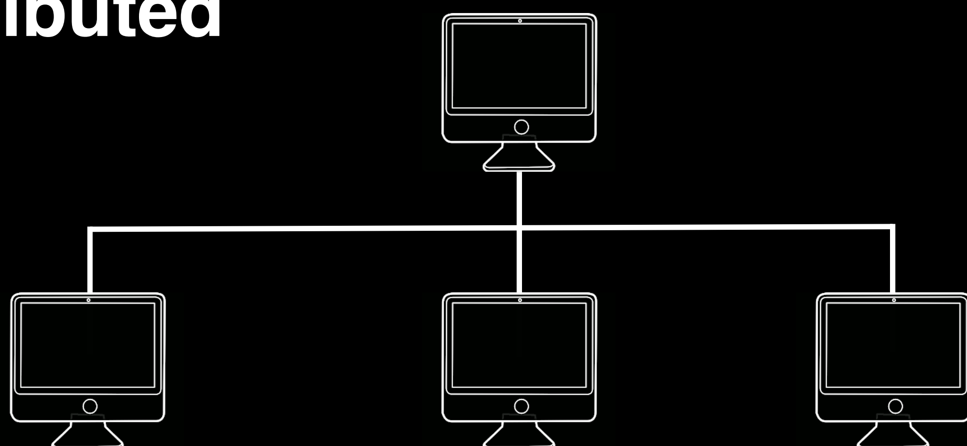
Morning Part 1
Goals

- Local vs Distributed
- Distributed Systems Architecture
- Hadoop File System & Hadoop MapReduce

Local vs Distributed

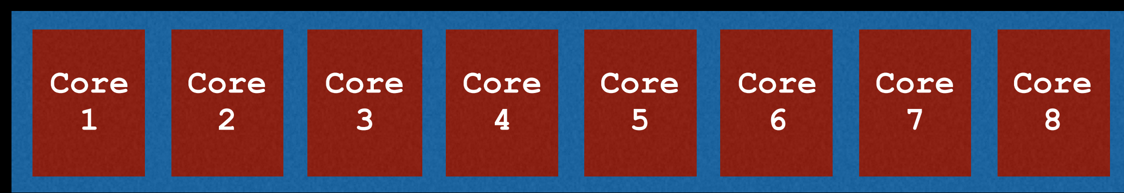
Local

Distributed

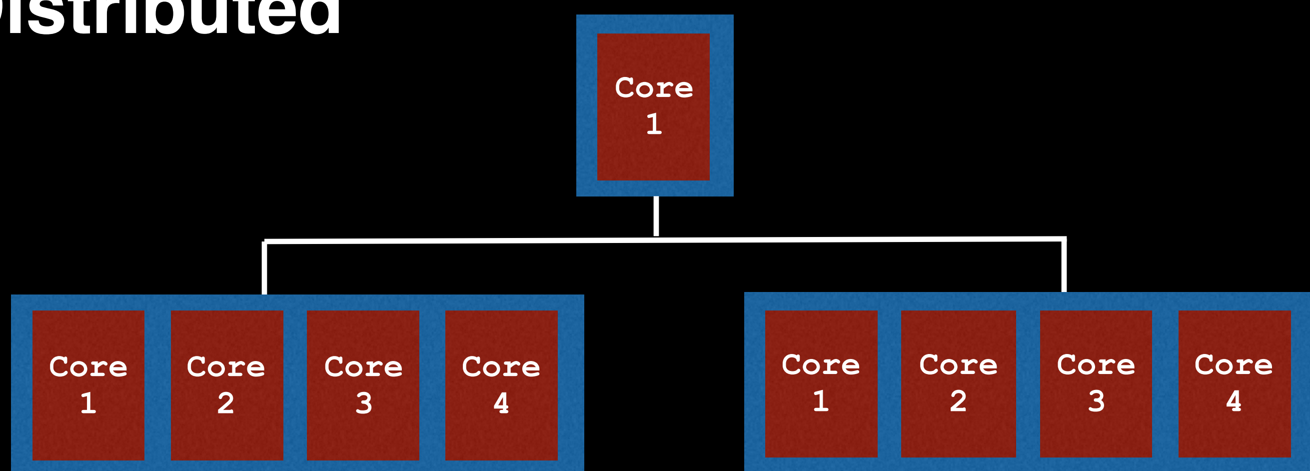


Local vs Distributed
(CPU)

Local



Distributed

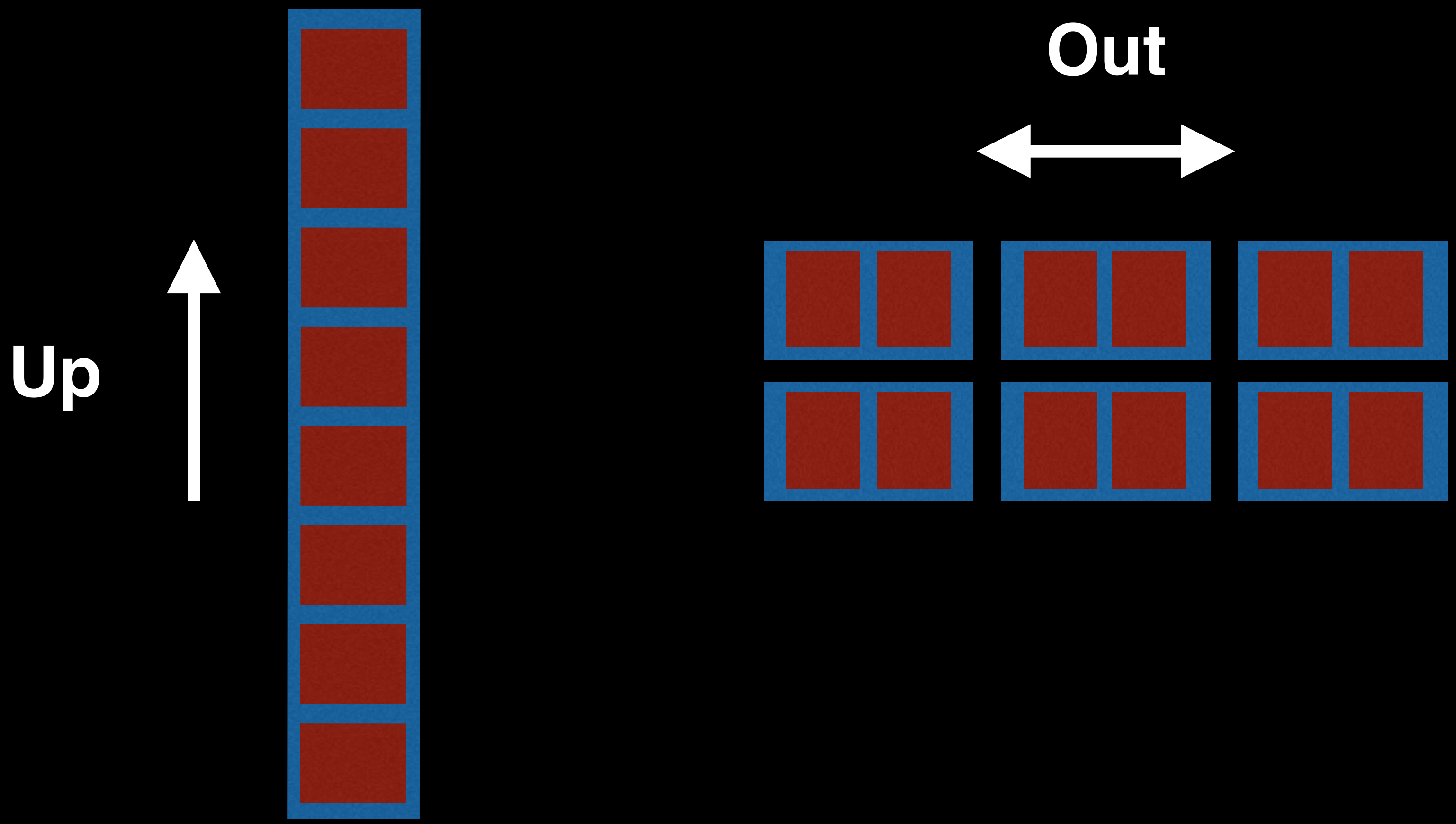


Local vs Distributed

- **Local processes**
  - ⋆ Use computation resources in 1 machine
- **Distributed processes**
  - ⋆ Use computation resources across a number of machines
  - ⋆ Connected via ethernet network (LAN)

Scaling

Up Vs Out



Local vs Distributed

- Local machines with a lot of CPU are expensive
- Many low spec. machines are cheaper

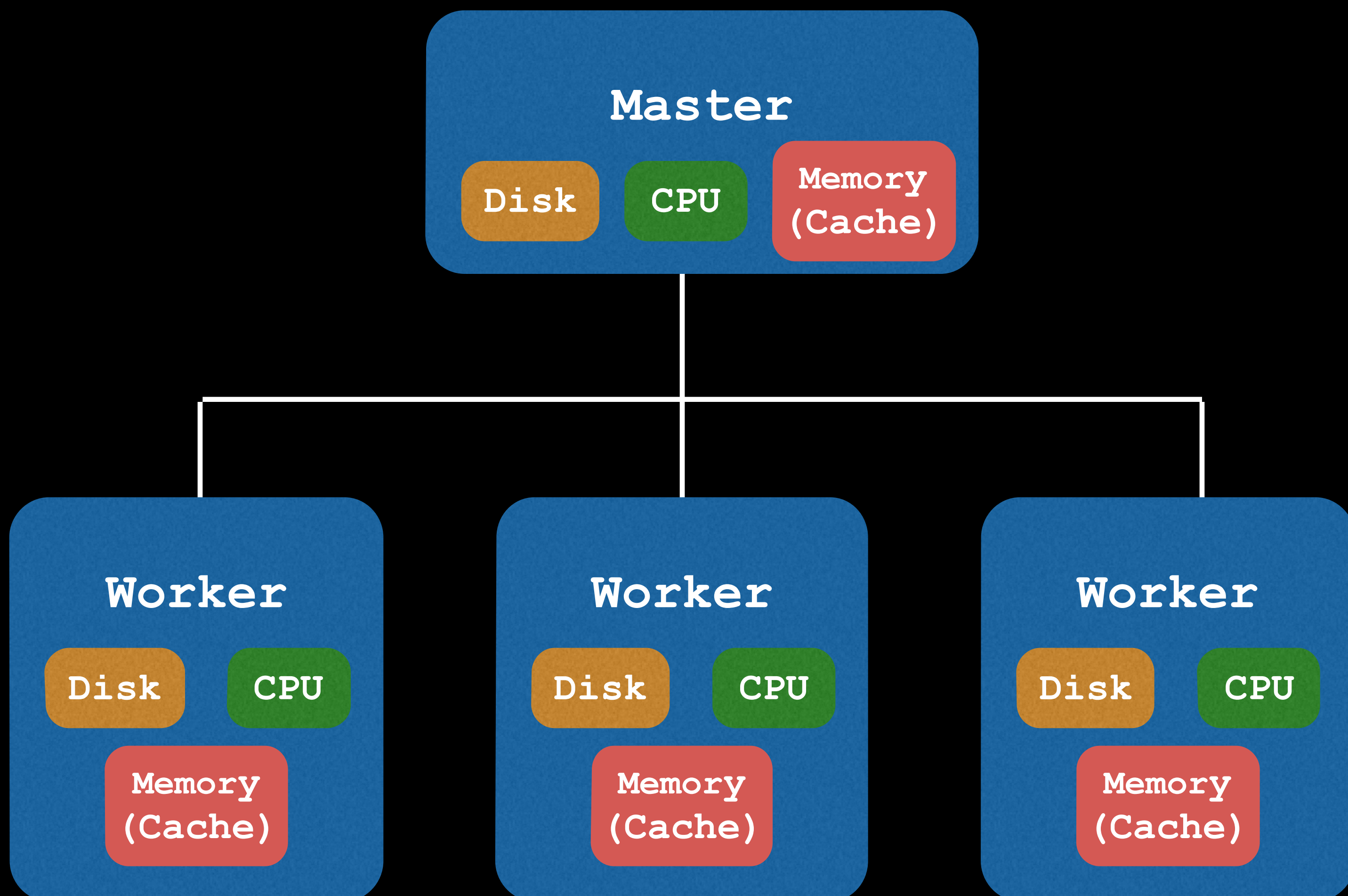
Local vs Distributed

- **Distributed: Better scalability**
  - ★ Easier to add more machines than to add more cores to a machine
- **Distributed: Fault tolerance**
  - ★ If one machine fails, the whole network is not down

Morning Part 1
Goals

- Local vs Distributed
- Distributed Systems Architecture
- Hadoop File System & Hadoop MapReduce

Distributed System Architecture



Morning Part 1
Goals

- Local vs Distributed
- Distributed Systems Architecture
- Hadoop File System & Hadoop MapReduce

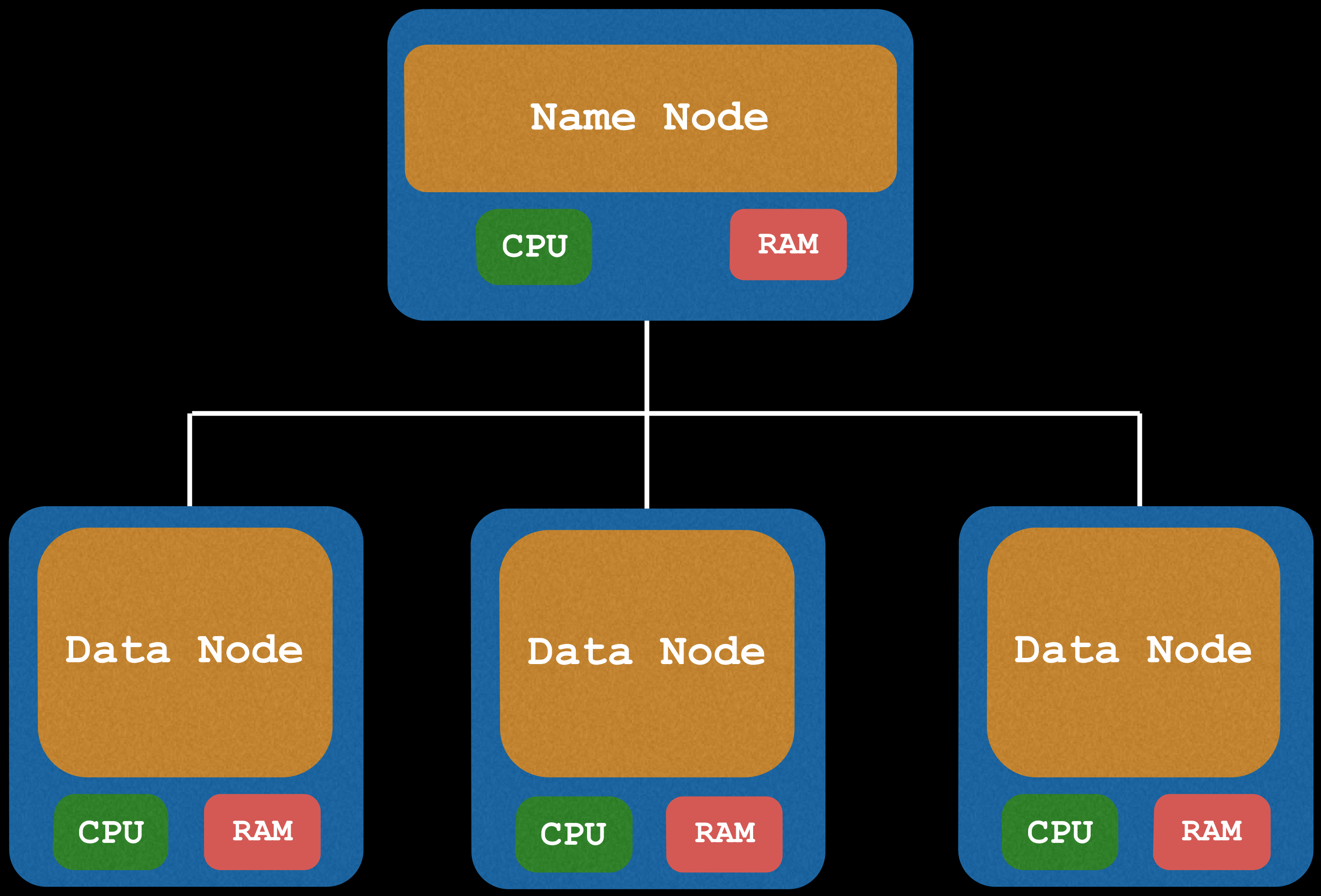
Hadoop Ecosystem

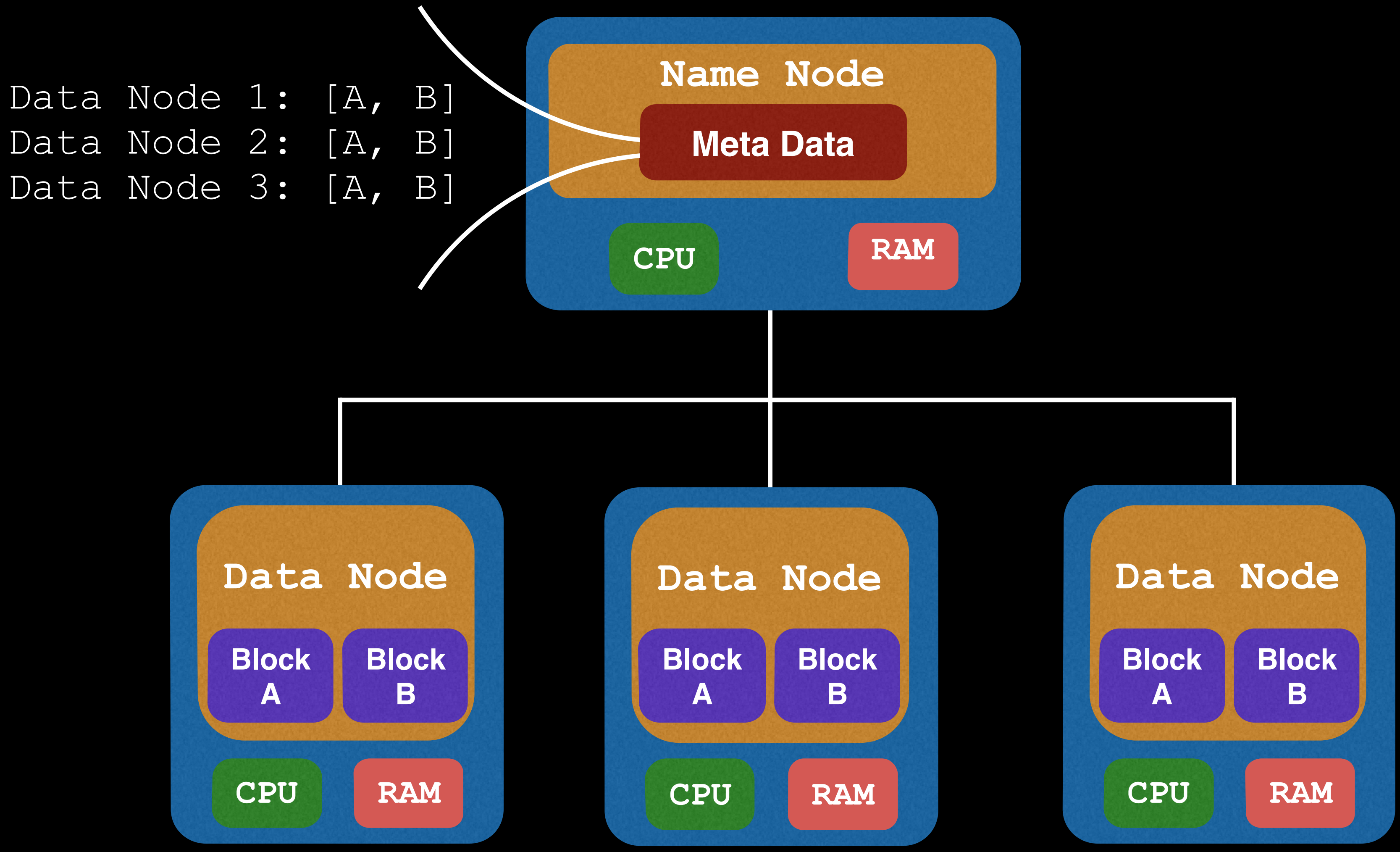
- **Distributed Storage**
  - HDFS (Hadoop File System)

- **Distributed processing**
  - MapReduce

Distributed Storage

HDFS



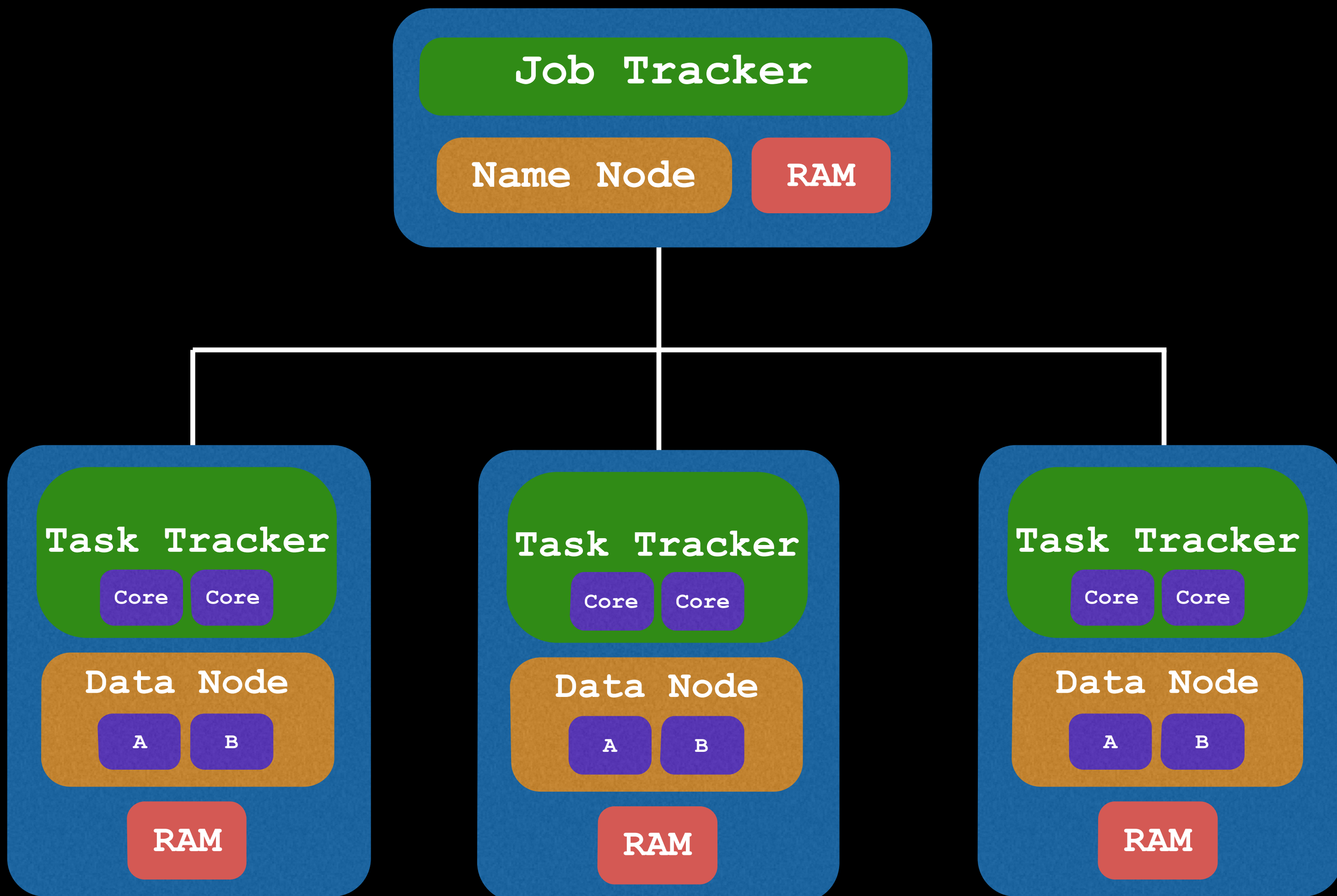


- Each block in 128 MB (By default)
- Each block is replicated 3 times

Blocks

- Smaller blocks provide more parallelization during processing
- But too small incurs penalty (Covered Later)
- Multiple copies of a block prevents loss of data due to failure of a node

Distributed Processing
MapReduce



MapReduce

- **Job Tracker**
  - Sends code to run on Task Trackers
- **Task Tracker**
  - Allocate CPU and memory for tasks
  - And monitor tasks on worker nodes

Break

Introduction to Spark

Morning Part 2
Goals

- Background
- Spark vs Hadoop MapReduce
- Spark Architecture / Basics
- Functional Programming
- Lazy Evaluation and Persistence
- Spark in Practice

Morning Part 2
Goals

- Background
- Spark vs Hadoop MapReduce
- Spark Architecture / Basics
- Functional Programming
- Lazy Evaluation and Persistence
- Spark in Practice

Background

- v1.3 (Released April 17th, 2015)
  - Added DataFrame capabilities

    (http://rustyrazorblade.com/2015/05/on-the-bleeding-edge-pyspark-dataframes-and-cassandra/)

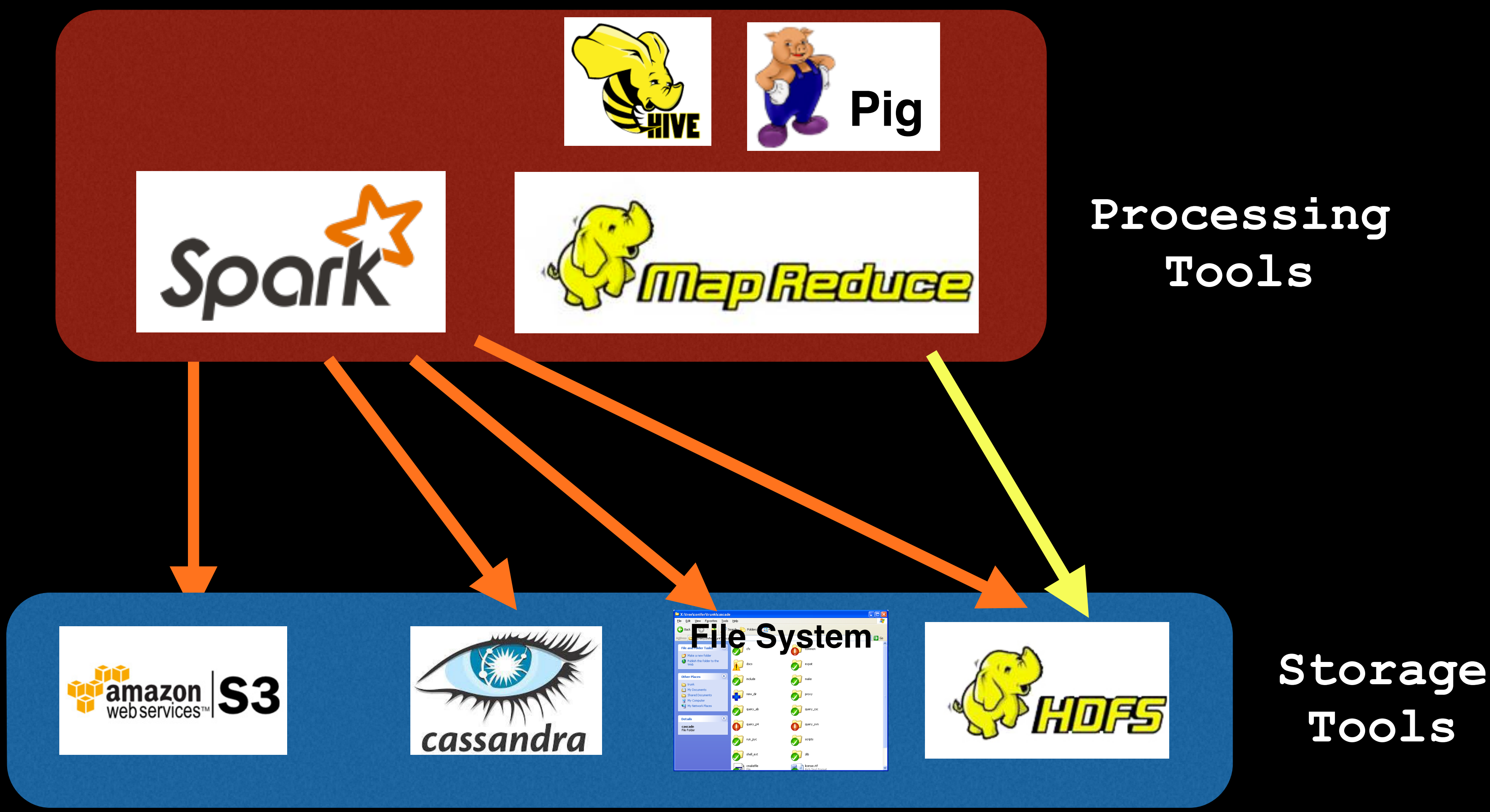
- First release (v0.7 on Feb, 2013)
- Found by AMPLab, UC Berkeley
- Now managed by DataBricks

Morning Part 2
Goals

- Background
- Spark vs Hadoop MapReduce
- Spark Architecture / Basics
- Functional Programming
- Lazy Evaluation and Persistence
- Spark in Practice

Spark vs MapReduce

Storage Compatibility



- **Spark**
  - ★ Does not require HDFS
- **MapReduce**
  - ★ Requires data to be loaded into HDFS

Spark vs MapReduce

Speed

Up to
100x faster than

- **Writes most data to disk** after each map and reduce operation

- **Keeps most data in memory** after each transformation
- Spill to disk if memory is filled
  (Only since v0.9)
- Before v0.9, would run out of memory

Spark vs MapReduce
Functionality

Spark

```
map()       groupByKey()
filter()    sortBy()
reduce()    join()
first()     count()

… and more …
```

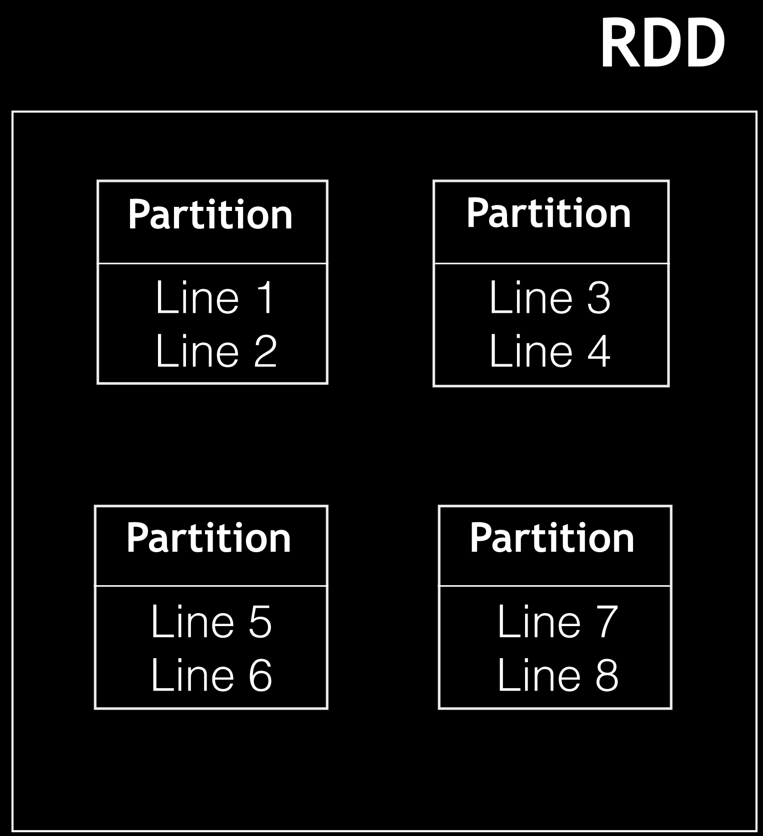
MapReduce

```
map()
reduce()
```

Morning Part 2
Goals

- Background
- Spark vs Hadoop MapReduce
- Spark Architecture / Basics
- Functional Programming
- Lazy Evaluation and Persistence
- Spark in Practice

Spark Core
Resilient Distributed Dataset

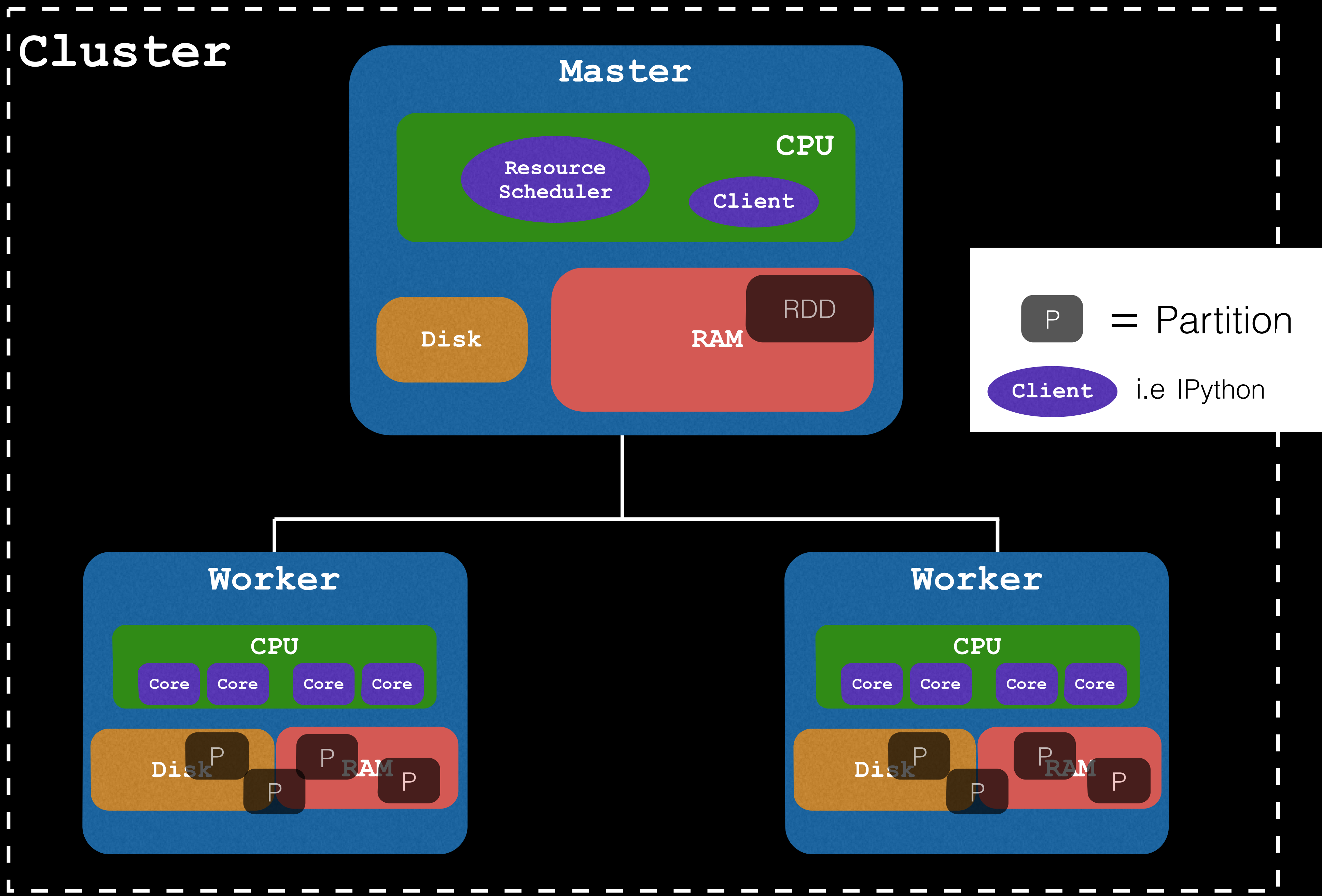


Number of Partitions

- By default, Spark has a default number of partitions
- **More partitions**
  - ★ More parallel processes
  - ★ More overhead in accessing

- **Choose k partitions based on**

```
Gain in Parallelization > Loss in overhead
```



Cluster
Master
CPU
Resource Scheduler
Client
Disk
RAM
RDD
Worker
CPU
Core
Core
Core
Core
Disk
RAM
P
P
P
P
Worker
CPU
Core
Core
Core
Core
Disk
RAM
P
P
P
P
P
= Partition
Client
i.e IPython

Spark Client / Spark Context

- **IPython / IPython Notebook** can be a **client** to interact with the master

- Client has a **SparkContext**

  - ★ A gateway between client and Spark

  - ★ To send code/data from IPython to Master (then to workers)

  - ★ Also can specify special settings of cluster(SparkConf)

Start a Cluster Locally

- The following starts a cluster with only the driver

```
import pyspark as ps

sc = ps.SparkContext('local[4]')
```

- Using all 4 cores (Similar to `multiprocessing`)

Start a Cluster Locally

- Start a master node

```
${SPARK_HOME}/bin/spark-class\
org.apache.spark.deploy.master.Master\
-h 127.0.0.1 \
-p 7077 \
--webui-port 8080
```

Domain to run master on

Port to run web UI on

Port to run master on

- Start worker node(s)

```
${SPARK_HOME}/bin/spark-class
org.apache.spark.deploy.worker.Worker
-c 1  \
-m 1G \
spark://127.0.0.1:7077
```

Worker assigned 1 core

Worker assigned 1G RAM

Master URI to link to

- Attach IPython to master node

```
IPYTHON_OPTS="notebook" \
${SPARK_HOME}/bin/pyspark \
--master spark://127.0.0.1:7077 \
—executor-memory 1G \
--driver-memory 1G
```

Master URI

Memory of each worker

Memory of driver

- SparkContext (`sc`) is loaded in by default

Starting a cluster locally

- **Simulate distributed systems** locally on 1 machine
- **For testing purposes** before we deploy the script to a distributed system

Create an RDD

Number of Partitions

- **Turn a list into an RDD**

```
rdd = sc.parallelize([1, 3, 4, 5, 6], n)
```

- **Read into RDD from file**

```
rdd = sc.textFile('path/to/file', n)
```

- **Read all files into RDD from a folder**

```
rdd = sc.textFile('path/to/folder', n)
```

Break

Morning Part 2
Goals

- Background
- Spark vs Hadoop MapReduce
- Spark Architecture / Basics
- Functional Programming
- Lazy Evaluation and Persistence
- Spark in Practice

Functional Programming

- **Apply a function to an RDD to create a new RDD**
  - RDDs are immutable
- Functions are passed to workers to and applied to partitions of the RDD

Functional Programming

- **By default items in list are (Key, Value items)**

Key **Value**

```
rdd = sc.parallelize([('a', 1), ('b', 2), ('a', 3)])
```

```
rdd2 = rdd.reduceByKey(lambda x, y: x + y)
```

```
rdd2.collect()
```

```
[('a', 4), ('b', 2)]
```

Two types of function

- **Transformation**
  - Return a new RDD
- **Action**
  - Return a final value
    (Python variable)

Two types of function

- **Transformation**

```
map()              groupByKey()
flatMap()          filter()
sortBy()           join()
reduceByKey()
```

- **Action**

```
first()            countByKey()
take()             collect()
count()            saveAsTextFile()
reduce()
```

Transformation

```
rdd = sc.parallelize(['apple orange', 'rainfall sunglass'])
```

```
rdd2 = rdd.flatMap(lambda item: item.split())
rdd3 = rdd.map(lambda item: item.split())
```

```
print rdd2.collect()
print rdd3.collect()
```

```
['apple', 'orange', 'rainfall', 'sunglass']
[['apple', 'orange'], ['rainfall', 'sunglass']]
```

- Action will load result onto Driver node
- **Do not use `collect()` on a large RDD**

Sampling RDD

- Often need to down-sample to RDD to test code
- **Fastest sampling**
  - `take(n)` will get the first rows
- **Fastest random sampling**
  - `sample(0.01, n)` will get **1%** rows at random

Save RDD to S3

- Saving to S3 is time consuming
- **Save to S3 only important / final results**

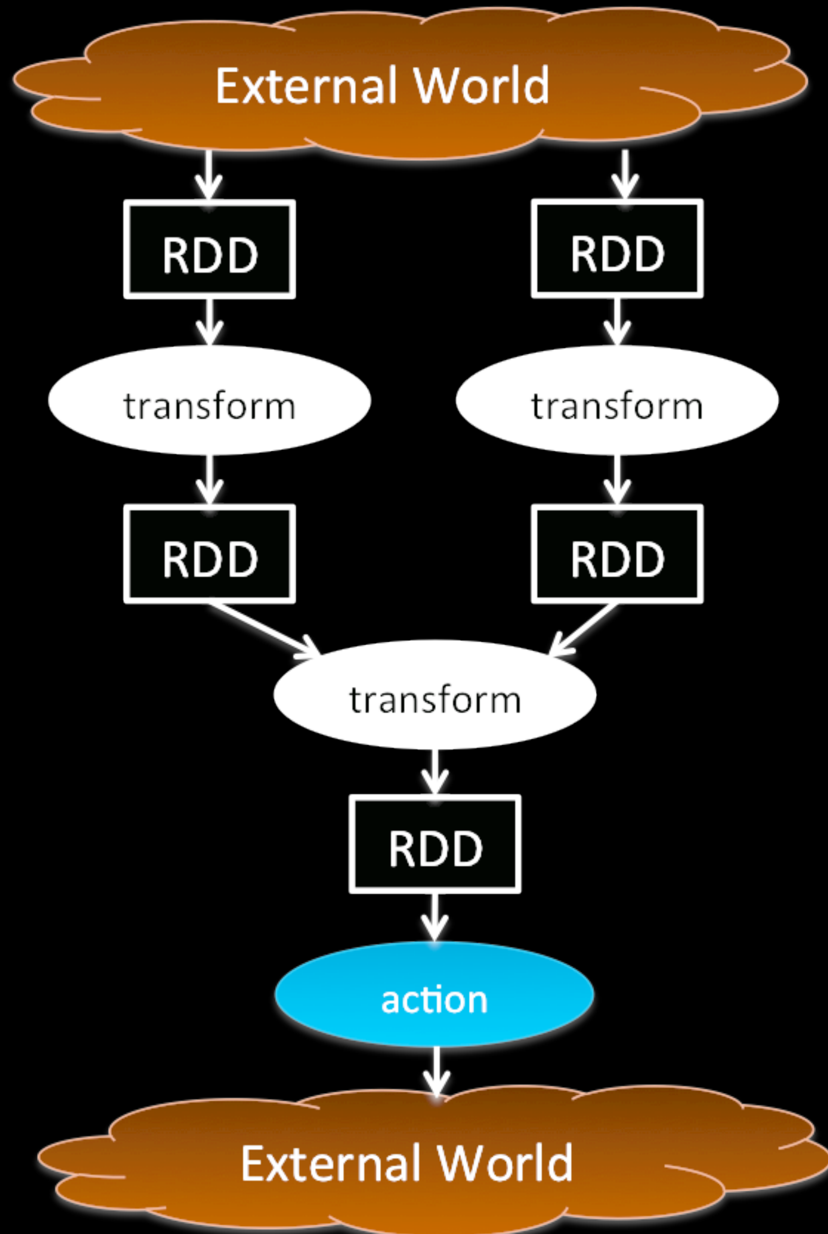
```
url = 's3://my-bucket-name/my-filename'

rdd.saveAsTextFile(url)
```

Upload | Create Folder | Actions | None | Properties | Transfers

All Buckets / jyt109

	--	--	--
block_-1955600302545797438	Standard	26.1 KB	Wed Jun 03 15:33:59 GMT-700 2015
block_-3474301327634414704	Standard	25.8 KB	Wed Jun 03 15:35:39 GMT-700 2015
block_-4480958310343221415	Standard	25.4 KB	Wed Jun 03 15:33:36 GMT-700 2015
block_-6046821718128745212	Standard	24.9 KB	Wed Jun 03 15:31:30 GMT-700 2015
block_1345845186870101660	Standard	26.5 KB	Wed Jun 03 15:31:12 GMT-700 2015
block_2531044467012315310	Standard	0 bytes	Wed Jun 03 15:28:35 GMT-700 2015
block_2673212514175746485	Standard	0 bytes	Wed Jun 03 15:35:54 GMT-700 2015
block_3881911621830791914	Standard	0 bytes	Wed Jun 03 15:29:34 GMT-700 2015
block_4181920493110957339	Standard	27.7 KB	Wed Jun 03 15:34:14 GMT-700 2015
block_4955688019551249909	Standard	23.2 KB	Wed Jun 03 15:33:44 GMT-700 2015
block_5182208915867884280	Standard	25.8 KB	Wed Jun 03 15:29:16 GMT-700 2015
block_5548308462198267834	Standard	23.8 KB	Wed Jun 03 15:28:50 GMT-700 2015
block_9123489716838483827	Standard	26 KB	Wed Jun 03 15:35:39 GMT-700 2015


External World
RDD
RDD
transform
transform
RDD
RDD
transform
RDD
action
External World

Along the series of transformations:

Try to keep the items of your RDD as `(key, value)`

Morning Part 2
Goals

- Background
- Spark vs Hadoop MapReduce
- Spark Architecture / Basics
- Functional Programming
- Lazy Evaluation and Persistence
- Spark in Practice

Lazy Evaluation

Very Important

- **Transformations are not run** when you run the command
- **Only actions** will cause the transformations previous to the action to run

Nothing runs here

```
rdd = sc.parallelize(range(1e10))

rdd2 = rdd.map(lambda x: x * 0.4)

rdd3 = rdd2.filter(lambda x: x > 1e4)


result = rdd3.reduce(lambda a, b: a + b)
```

Everything runs here

Caching

- **Explicitly keep an rdd in memory**
- **Only if:**
    - RDD used for different operations many times (Faster)

```
rdd.setName(name)

rdd.persist()
```

Types of Caching

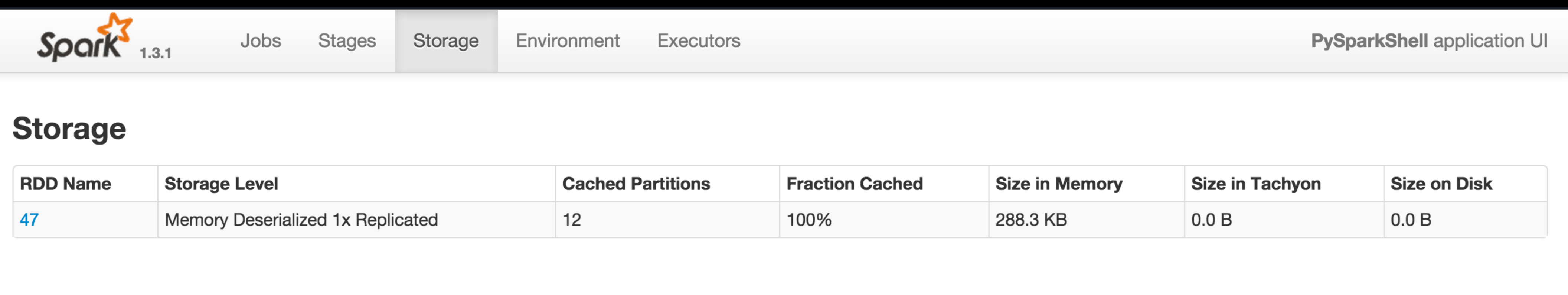
```
from pyspark.storagelevel import StorageLevel

rdd.setName(name)
rdd.persist(StorageLevel.MEMORY_ONLY)
```

Storage Level	**Meaning**
MEMORY_ONLY	Store RDD as deserialized Java objects in the JVM. If the RDD does not fit in memory, some partitions will not be cached and will be recomputed on the fly each time they're needed. This is the default level.
MEMORY_AND_DISK	Store RDD as deserialized Java objects in the JVM. If the RDD does not fit in memory, store the partitions that don't fit on disk, and read them from there when they're needed.
MEMORY_ONLY_SER	Store RDD as *serialized* Java objects (one byte array per partition). This is generally more space-efficient than deserialized objects, especially when using a fast serializer, but more CPU-intensive to read.
MEMORY_AND_DISK_SER	Similar to MEMORY_ONLY_SER, but spill partitions that don't fit in memory to disk instead of recomputing them on the fly each time they're needed.
DISK_ONLY	Store the RDD partitions only on disk.

- Caching will only take place **when an action is executed** (lazy evaluation)

- Can also do **`rdd.unpersist()`** to free memory



Storage

RDD Name	Storage Level	Cached Partitions	Fraction Cached	Size in Memory	Size in Tachyon	Size on Disk
47	Memory Deserialized 1x Replicated	12	100%	288.3 KB	0.0 B	0.0 B

Morning Part 2
Goals

- Background
- Spark vs Hadoop MapReduce
- Spark Architecture / Basics
- Functional Programming
- Lazy Evaluation and Persistence
- Spark in Practice

Spark In Practice

- Set up master / workers
- Open up UI at **`domain:8080`** (e.g. **`localhost:8080`**)
- Attach IPython client to master
- Load data in and start data manipulation
- Track UI as you execute your commands

Spark UI

Application Page

More detailed guide in the repo

Spark Master at spark://127.0.0.1:7077

Spark 1.3.1

URL: spark://127.0.0.1:7077
REST URL: spark://127.0.0.1:6066 *(cluster mode)*
Workers: 2
Cores: 2 Total, 2 Used
Memory: 2.0 GB Total, 2.0 GB Used
Applications: 1 Running, 0 Completed
Drivers: 0 Running, 0 Completed
Status: ALIVE

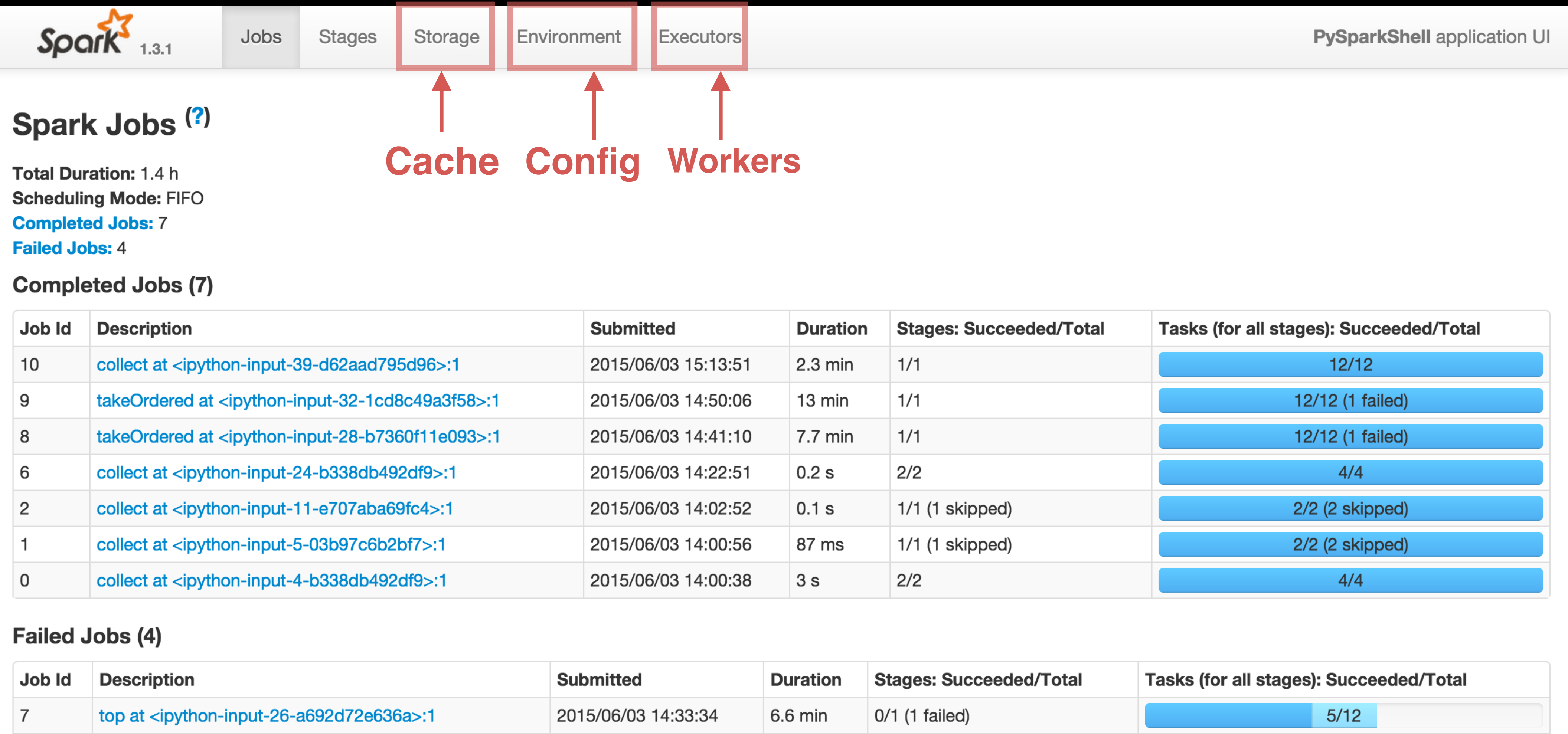
Workers

Worker Id	Address	State	Cores	Memory
worker-20150603135907-10.3.35.25-56877	10.3.35.25:56877	ALIVE	1 (1 Used)	1024.0 MB (1024.0 MB Used)
worker-20150603135911-10.3.35.25-56879	10.3.35.25:56879	ALIVE	1 (1 Used)	1024.0 MB (1024.0 MB Used)

Running Applications

Application ID	Name	Cores	Memory per Node	Submitted Time	User	State	Duration
app-20150603135926-0000	PySparkShell	2	1024.0 MB	2015/06/03 13:59:26	jeffreytang	RUNNING	1.4 h

Spark UI
Jobs Page



Spark 1.3.1 | Jobs | Stages | Storage | Environment | Executors | **PySparkShell** application UI

Cache Config Workers

Spark Jobs (?)

Total Duration: 1.4 h
Scheduling Mode: FIFO
Completed Jobs: 7
Failed Jobs: 4

Completed Jobs (7)

Job Id	Description	Submitted	Duration	Stages: Succeeded/Total	Tasks (for all stages): Succeeded/Total
10	collect at <ipython-input-39-d62aad795d96>:1	2015/06/03 15:13:51	2.3 min	1/1	12/12
9	takeOrdered at <ipython-input-32-1cd8c49a3f58>:1	2015/06/03 14:50:06	13 min	1/1	12/12 (1 failed)
8	takeOrdered at <ipython-input-28-b7360f11e093>:1	2015/06/03 14:41:10	7.7 min	1/1	12/12 (1 failed)
6	collect at <ipython-input-24-b338db492df9>:1	2015/06/03 14:22:51	0.2 s	2/2	4/4
2	collect at <ipython-input-11-e707aba69fc4>:1	2015/06/03 14:02:52	0.1 s	1/1 (1 skipped)	2/2 (2 skipped)
1	collect at <ipython-input-5-03b97c6b2bf7>:1	2015/06/03 14:00:56	87 ms	1/1 (1 skipped)	2/2 (2 skipped)
0	collect at <ipython-input-4-b338db492df9>:1	2015/06/03 14:00:38	3 s	2/2	4/4

Failed Jobs (4)

Job Id	Description	Submitted	Duration	Stages: Succeeded/Total	Tasks (for all stages): Succeeded/Total
7	top at <ipython-input-26-a692d72e636a>:1	2015/06/03 14:33:34	6.6 min	0/1 (1 failed)	5/12