Data Analysis with Hadoop & Spark

Part Two

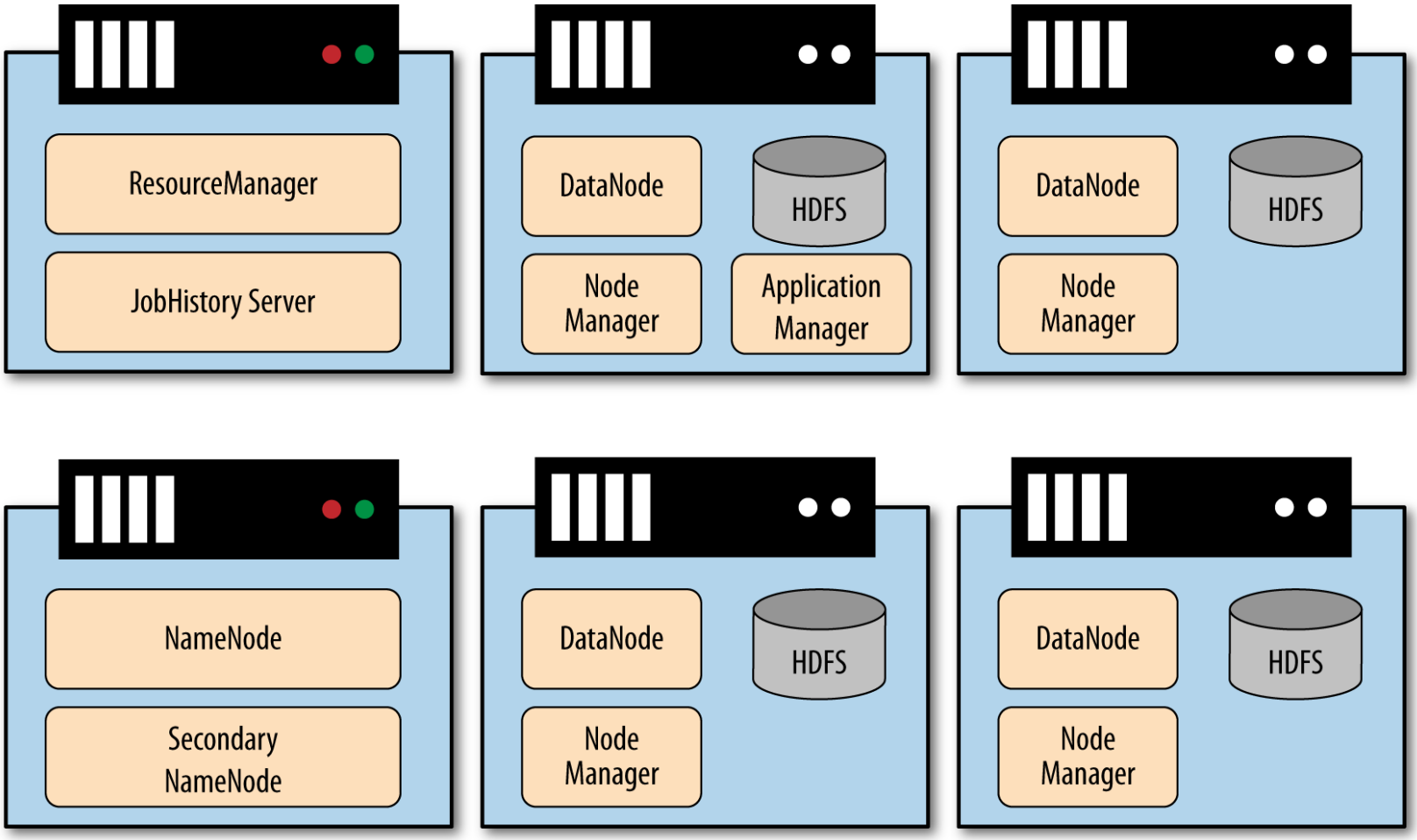
Agenda

- Setting Up for Big Data Analytics
  - Parts of a Hadoop Cluster
  - Parts of a Spark Cluster
- Introduction to Spark
  - Building Applications for Spark
  - Writing Spark Applications

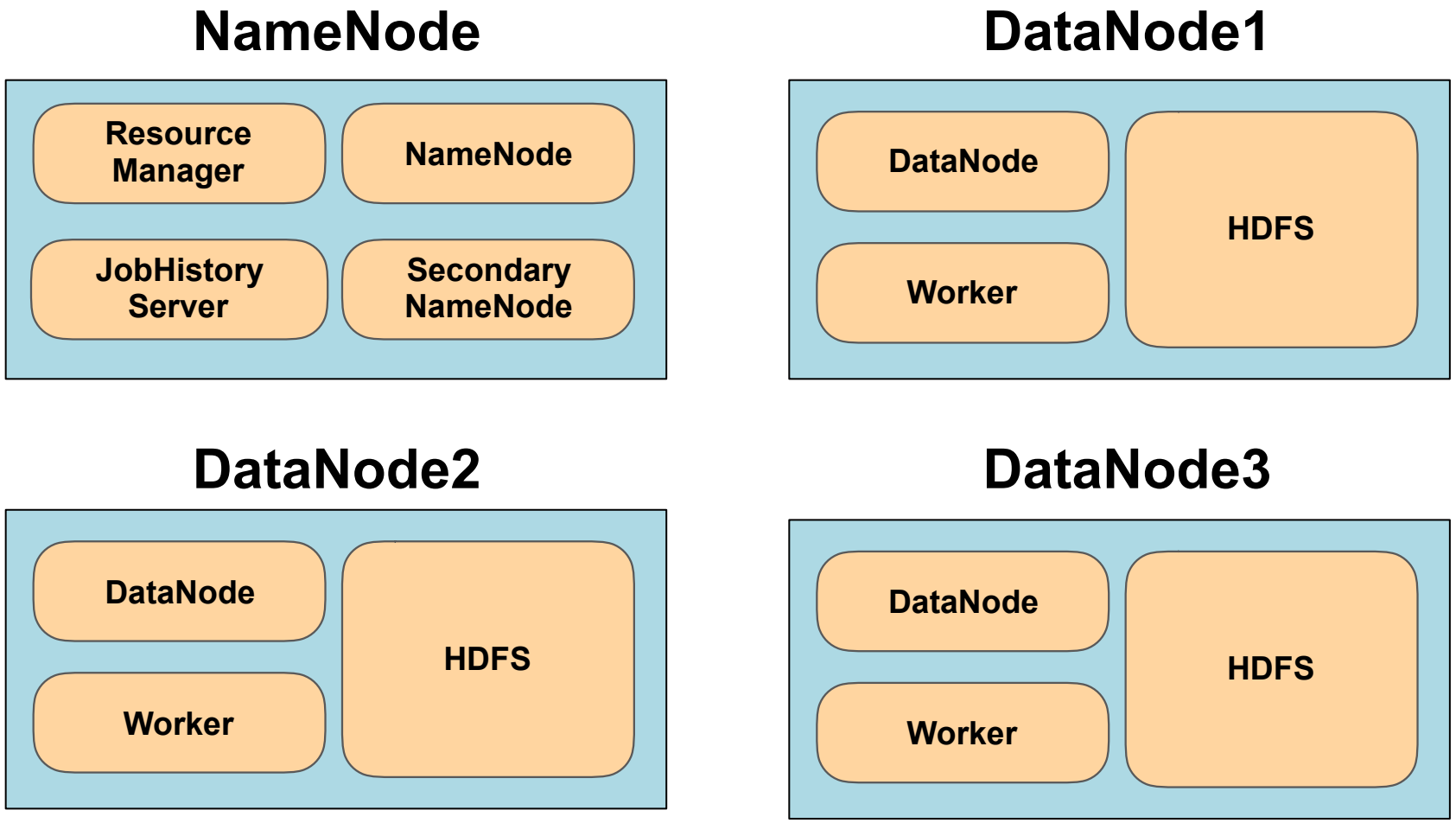
Setting Up for Big Data Analytics

Building a Hadoop Cluster in AWS

Cluster Example

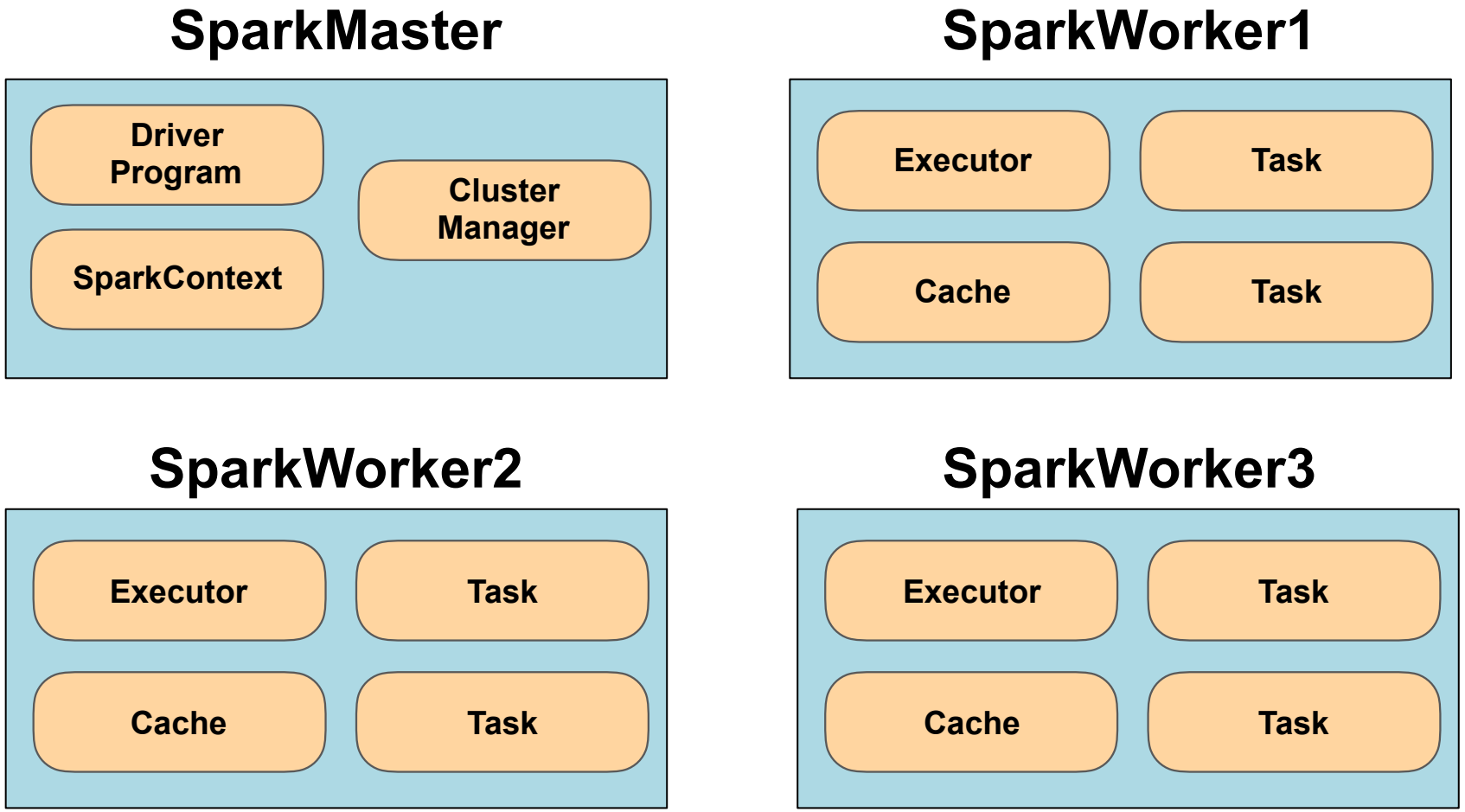


Our Hadoop Cluster



Building a Spark Cluster in AWS

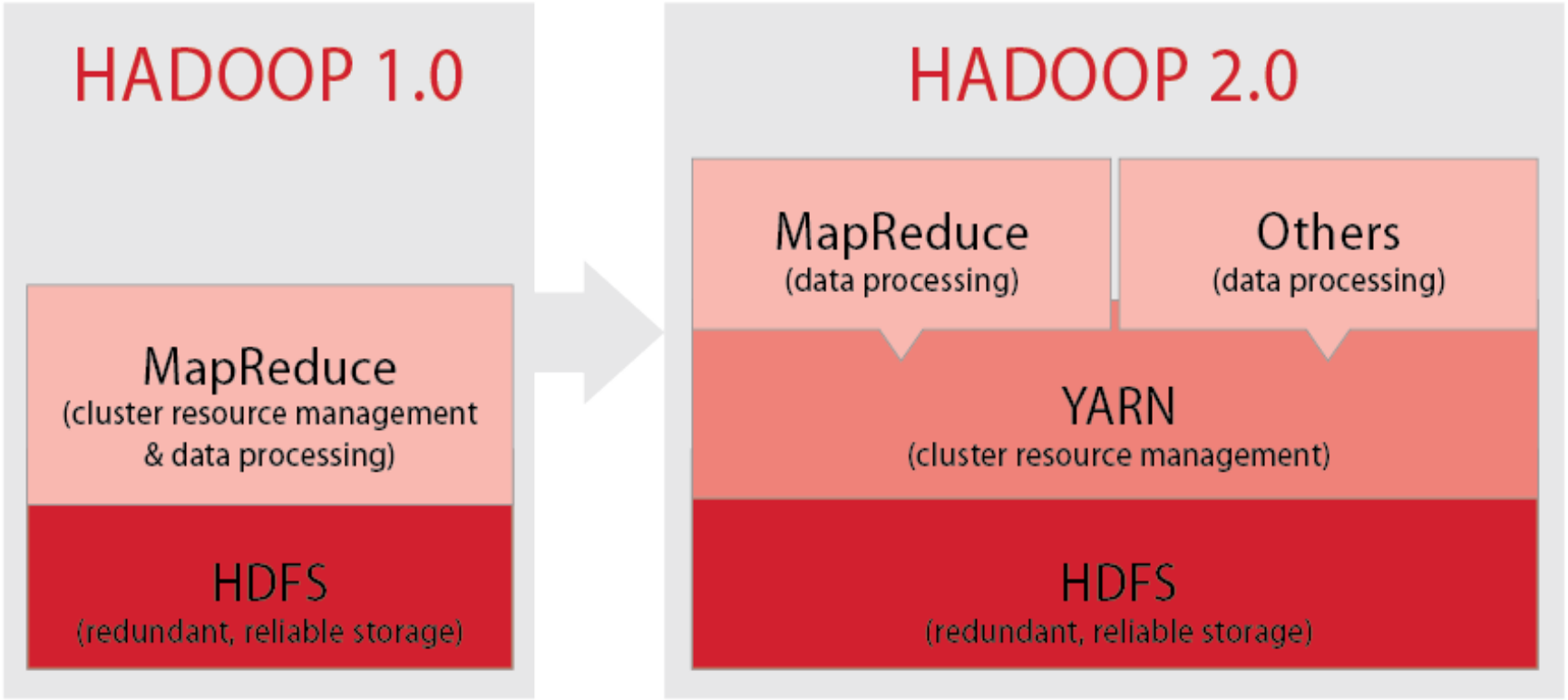
Our Spark Cluster



Introduction to

Building Applications with Spark

Evolution of Hadoop



Introducing Spark

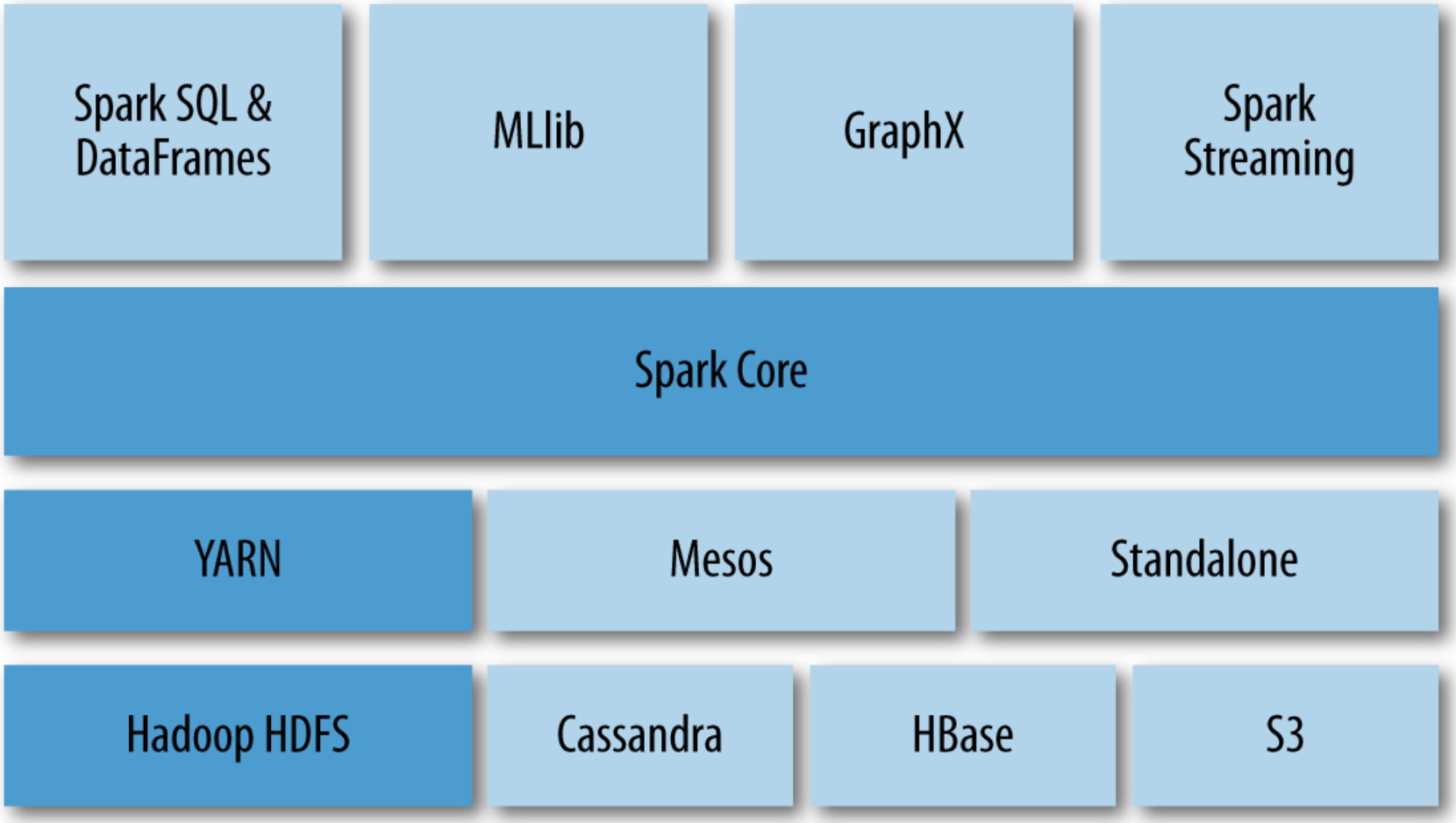
First fast, general-purpose paradigm resulting from this shift.

Achieves speed via new data model called **Resilient Distributed Datasets (RDDs)**.

Directed Acyclic Graph (DAG) execution engine that optimizes iterative computation.

Can be accessed in an interactive fashion, bringing the cluster to the data scientist.

The Spark Stack



The Spark Stack

Spark SQL - Provides DataFrames, collections of data organized similarly to tables in relational databases.

Spark Streaming - Processing and manipulation of unbounded streams of data in real time.

MLlib - Library of common machine learning algorithms implemented as Spark operations on RDDs.

GraphX - Collection of algorithms and tools for manipulating graphs and performing parallel graph operations and computations.

Resilient Distributed Datasets (RDDs)

Programming abstraction that represents read-only collection of objects partitioned across machines.

RDDs can be:

- Rebuilt from lineage (fault tolerant).
- Accessed via MapReduce-like (functional) parallel operations.
- Cached in memory for immediate reuse.
- Written to distributed storage.

These properties of RDDs all meet the Hadoop requirements for a distributed computation framework.

Programming with RDDs

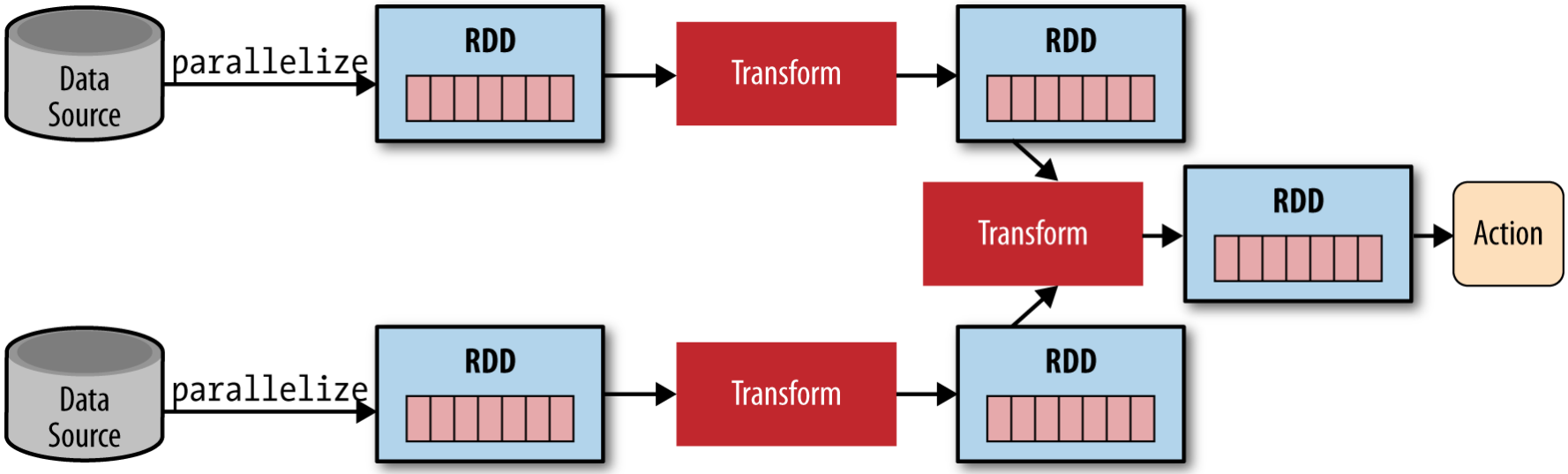
Code written in a driver program that is evaluated lazily and distributed across cluster to be executed by workers on their partitions of the RDD.

Results sent back to driver for aggregation and compilation.

Driver program creates one or more RDDs by parallelizing dataset from Hadoop data source.

Applies operations to transform the RDD and invokes some action on the RDD to retrieve output.

Example Spark Application



Typical Spark Data Flow

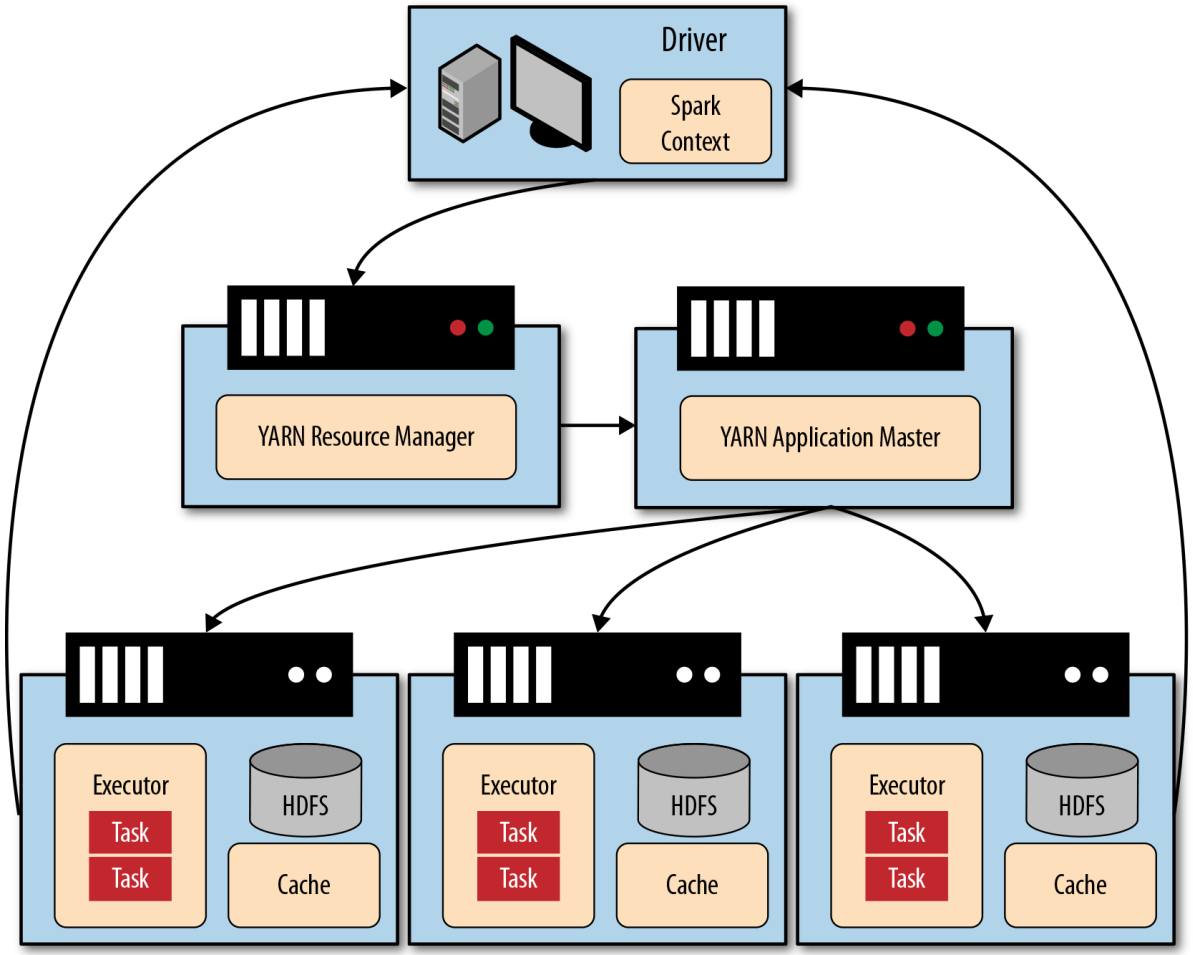
Define one or more RDDs, either through:

- Accessing data stored on disk
- Parallelizing some collection
- Transforming existing RDD
- Caching.

Invoke operations on RDD by passing closures to each element.

Use resulting RDDs with aggregating actions.

Spark Execution Model



Working with RDDs

Most people focus on the in-memory caching of RDDs, which is great because it allows for:

- Batch analyses (like MapReduce)
- Interactive analyses (humans exploring Big Data)
- Iterative analyses (no expensive Disk I/O)
- Real time processing (just “append” to the collection)

However, RDDs also provide a more general interaction with functional constructs at a higher level of abstraction: not just MapReduce!

Lab: Flight Delays

Spark

Spark Metrics

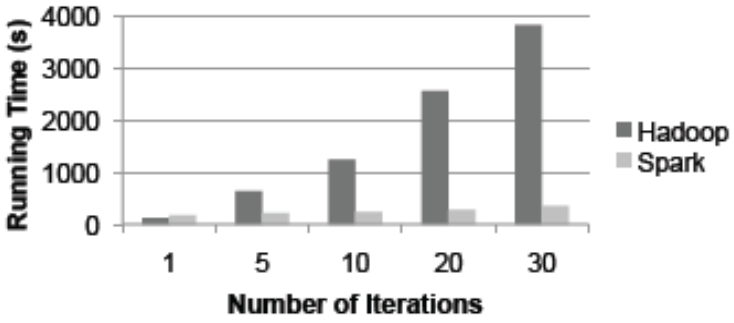
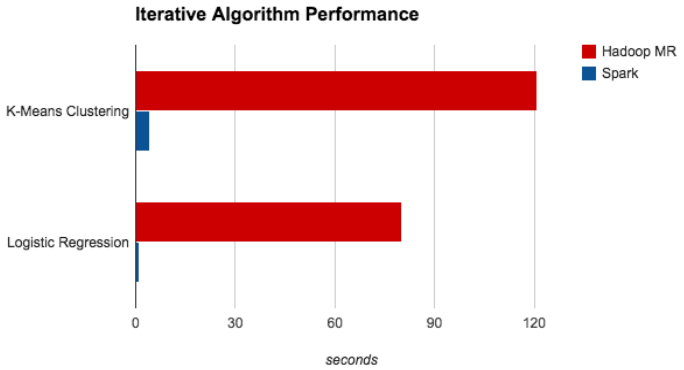
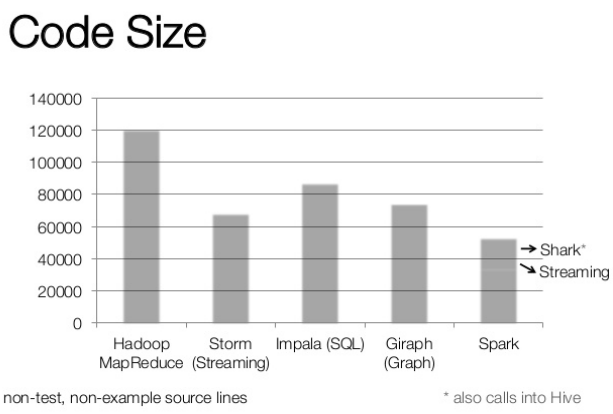


Figure 2: Logistic regression performance in Hadoop and Spark.





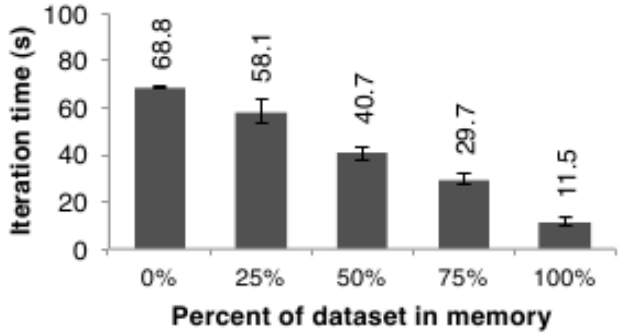


Figure 12: Performance of logistic regression using 100 GB data on 25 machines with varying amounts of data in memory.

Programming Spark

Create a driver program (app.py) that does the following:

1. Define one or more RDDs either through accessing data stored on disk.
2. Invoke operations on the RDD by passing closures (functions) to each element of the RDD.
3. Use the resulting RDDs with actions (ex. count, collect, save, etc.).

More details on this soon!

Spark Execution

Spark applications are run as independent sets of processes

Coordination is by a SparkContext in a driver program.

The context connects to a cluster manager which allocates computational resources.

Spark then acquires executors on individual nodes on the cluster.

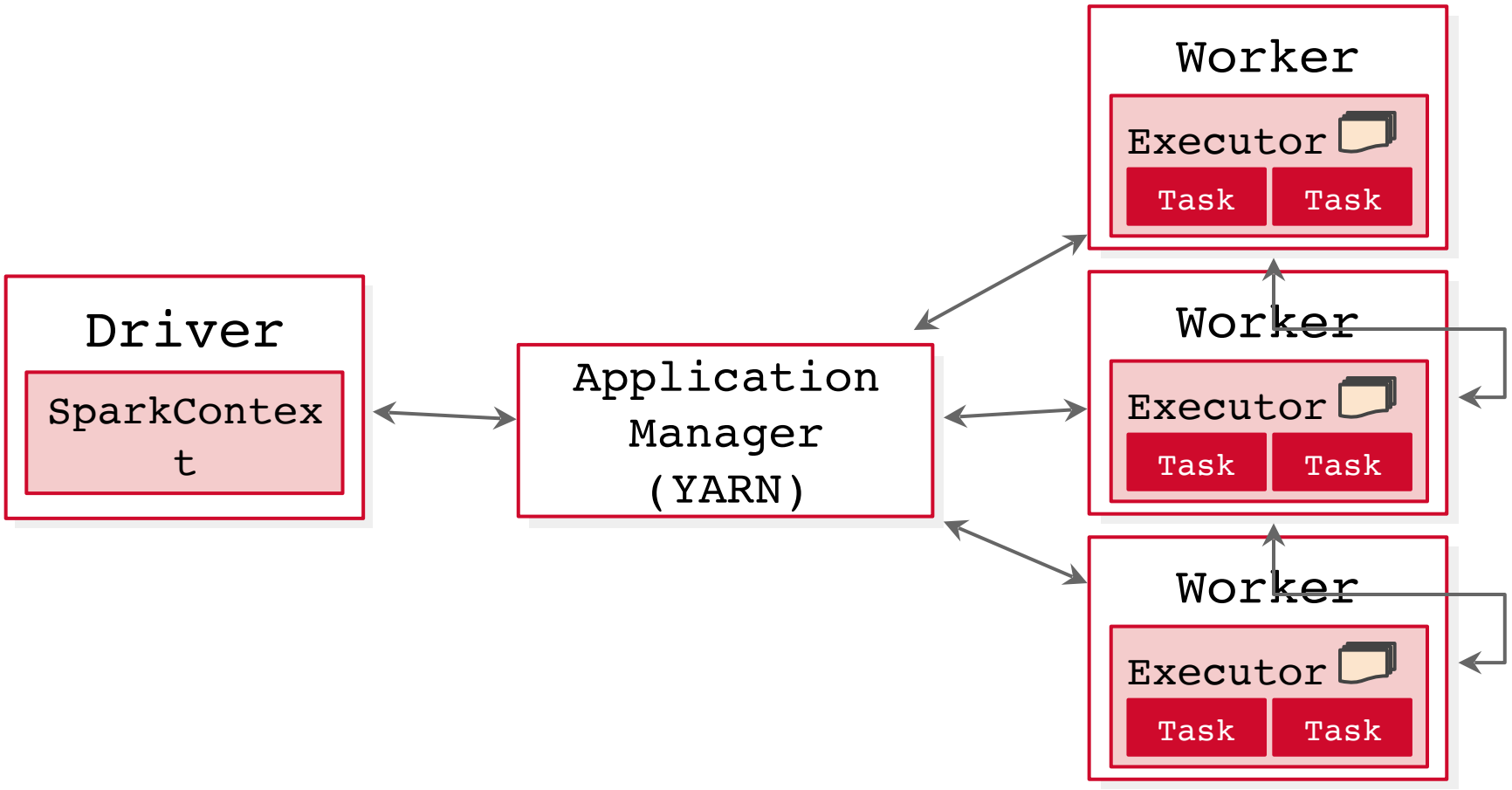
Spark Execution

Executors manage individual worker computations as well as manage the storage and caching of data.

Application code is sent from the driver to the executors which specifies the context and the tasks to be run.

Communication can occur between workers and from the driver to the worker.

Spark Execution



Key Points Regarding Execution

Each application gets its own executor for the duration, and tasks run in multiple threads or processes.

Data can be shared between executors, but not between different Spark applications without external storage.

Application Manager can be anything - YARN on Hadoop, Mesos, or Spark Standalone.

Drivers should be on the same local network with the cluster, and remote cluster access should use RPC access to driver.

Executing Spark Jobs

Use spark-submit to send your application to the cluster for execution with any other Python files and dependencies.

```
export HADOOP_CONF_DIR=XXX
/srv/spark/bin/spark-submit \
  --master yarn-cluster \
  --executor-memory 20G \
  --num-executors 50    \
  --py-files mydeps.egg
  App.py
```

This will cause Spark to allow the driver program to acquire a Context that utilizes the YARN ResourceManager.

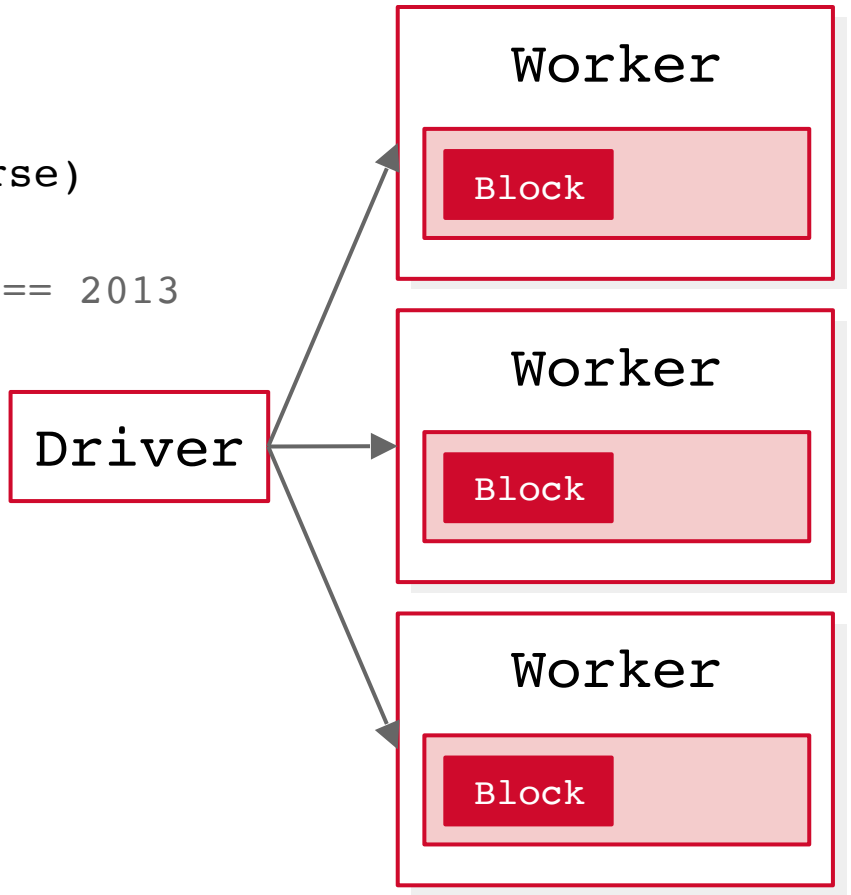
Spark Master URL

Master URL	Meaning
local	Run Spark locally with one worker thread (i.e. no parallelism at all).
local[K]	Run Spark locally with K worker threads (ideally, set this to the number of cores on your machine).
local[*]	Run Spark locally with as many worker threads as logical cores on your machine.
spark://HOST:PORT	Connect to the given Spark standalone cluster master. The port must be whichever one your master is configured to use, which is 7077 by default.
mesos:// HOST:PORT	Connect to the given Mesos cluster. The port must be whichever one your is configured to use, which is 5050 by default. Or, for a Mesos cluster using ZooKeeper, use mesos://zk://....
yarn-client	Connect to a YARN cluster in client mode. The cluster location will be found based on the HADOOP_CONF_DIR variable.
yarn-cluster	Connect to a YARN cluster in cluster mode. The cluster location will be found based on HADOOP_CONF_DIR.

Example Data Flow

```
# Base RDD
orders = sc.textFile("hdfs://...")
orders = orders.map(split).map(parse)
orders = orders.filter(
    lambda order: order.date.year == 2013
)
orders.cache()
```

1. Read Block from HDFS
2. Process RDDs
3. Cache the data in memory

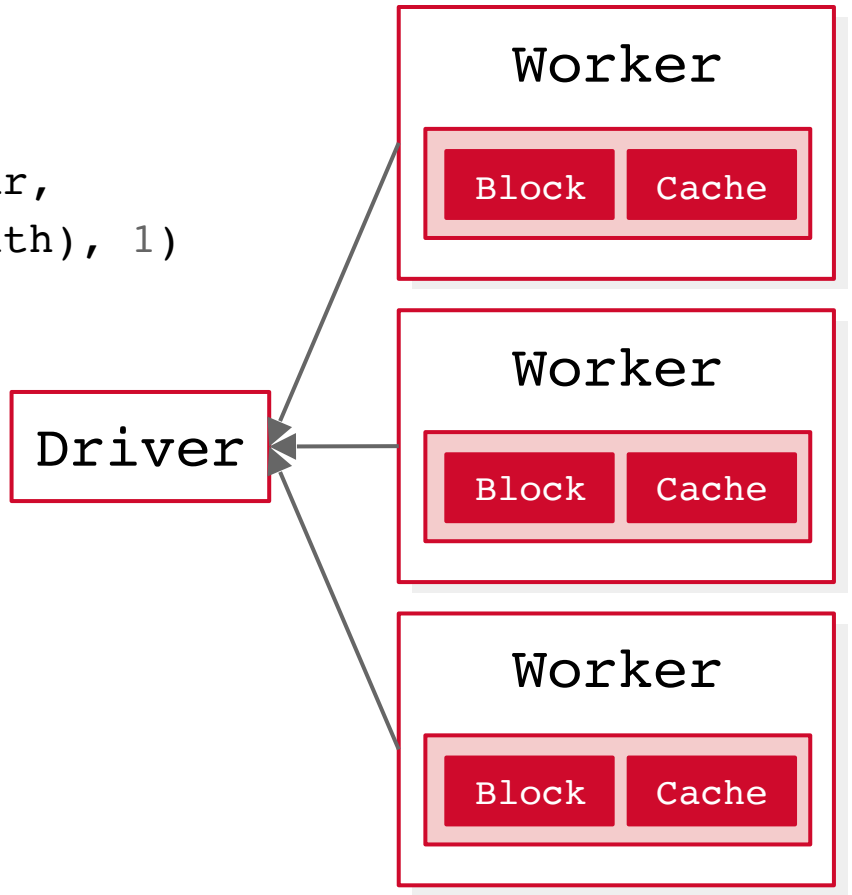


Example Data Flow

```
months = orders.map(
    lambda order: ((order.date.year,
                    order.date.month), 1)
    )

months = months.reduceByKey(add)
print months.take(5)
```

1. Process final RDD
2. On action send result back to driver
3. Driver outputs result (print)

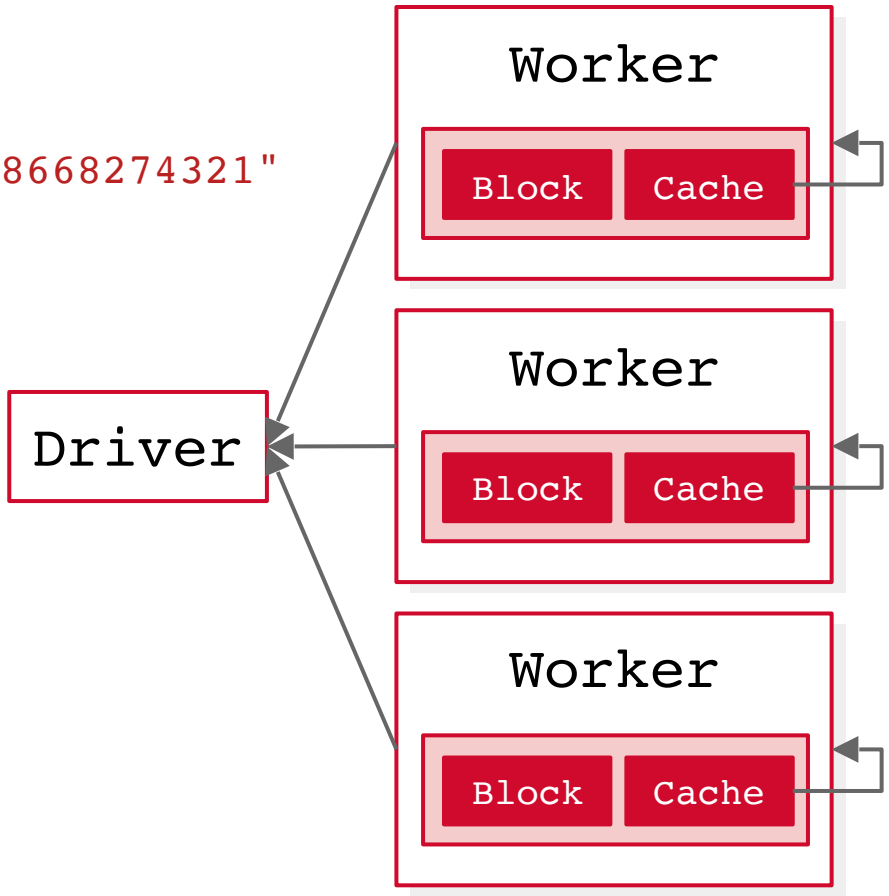


Example Data Flow

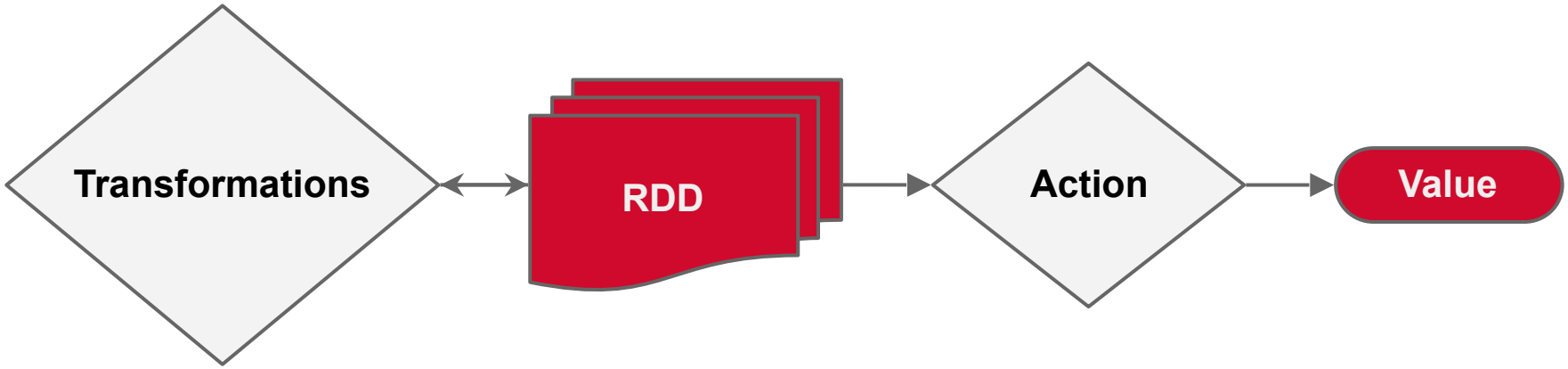
```
products = orders.filter(
    lambda order: order.upc == "098668274321"
)

print products.count()
```

1. Process RDD from cache
2. Send data on action back to driver
3. Driver outputs result (print)



Spark Data Flow



Debugging Data Flow

```
>>> print months.toDebugString()

(9) PythonRDD[9] at RDD at PythonRDD.scala:43
 |  MappedRDD[8] at values at NativeMethodAccessorImpl.java:-2
 |  ShuffledRDD[7] at partitionBy at
NativeMethodAccessorImpl.java:-2
 +-(9) PairwiseRDD[6] at RDD at PythonRDD.scala:261
    |  PythonRDD[5] at RDD at PythonRDD.scala:43
    |  PythonRDD[2] at RDD at PythonRDD.scala:43
    |  orders.csv MappedRDD[1] at textFile
    |  orders.csv HadoopRDD[0] at textFile
```

Operator Graphs and Lineage can be shown with the `toDebugString` method, allowing a visual inspection of what is happening under the hood.

Writing Spark Applications

Creating a Spark Application

Writing a Spark application in Java, Scala, or Python is similar to using the interactive console - the API is the same.

All you need to do first is to get access to the SparkContext that was loaded automatically for you by the interpreter.

```
from pyspark import SparkConf, SparkContext
conf = SparkConf().setAppName("MyApp")
sc = SparkContext(conf=conf)
```

To shut down Spark:

```
sc.stop() or sys.exit(0)
```

Structure of a Spark Application

Dependencies (import) - Third party dependencies can be shipped with app.

Constants and Structures - Especially namedtuples and other constants.

Closures - Functions that operate on the RDD.

A main method:

- Creates a SparkContext
- Creates one or more RDDs
- Applies *transformations* to RDDs
- Applies *actions* to kick off computation

Spark Application Skeleton

```
from pyspark import SparkConf, SparkContext

## Module Constants
APP_NAME = "My Spark Application"

## Closure Functions

## Main functionality
def main(sc):
    pass

if __name__ == "__main__":
    # Configure Spark
    conf = SparkConf().setAppName(APP_NAME)
    sc   = SparkContext(conf=conf)

    # Execute Main functionality
    main(sc)
```

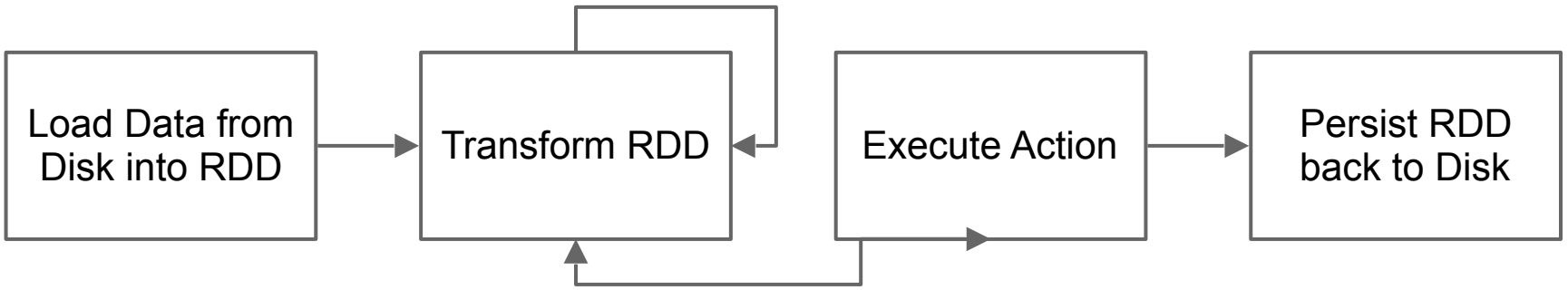
Programming Model

Two types of operations on an RDD:

- Transformations
- Actions

Transformations are lazily evaluated - they aren't executed when you issue the command.

RDDs are recomputed when an action is executed.



Initializing an RDD

Two types of RDDs:

Parallelized collections - take an existing in memory collection (a list or tuple) and run functions upon it in parallel.

Hadoop datasets - run functions in parallel on any storage system supported by Hadoop (HDFS, S3, HBase, local file system, etc).

Input can be text, `SequenceFiles`, and any other Hadoop `InputFormat` that exists.

Initializing an RDD

```
# Parallelize a list of numbers
distributed_data = sc.parallelize(xrange(100000))

# Load data from a single text file on disk
lines = sc.textFile('tolstoy.txt')

# Load data from all csv files in a directory using glob
files  = sc.wholeTextFiles('dataset/*.csv')

# Load data from S3
data = sc.textFile('s3://databucket/')
```

For HBase example, see: hbase_inputformat.py

Transformations

- Create a new dataset from an existing one.
- Evaluated lazily, won't be executed until action.

Transformation	Description
`map(func)`	Return a new distributed dataset formed by passing each element of the source through a function func.
`filter(func)`	Return a new dataset formed by selecting those elements of the source on which func returns true.
`flatMap(func)`	Similar to map, but each input item can be mapped to 0 or more output items (so func should return a Seq rather than a single item).

Transformations

Transformation	Description
`mapPartitions(func)`	Similar to map, but runs separately on each partition (block) of the RDD, so func must be of type Iterator<T> => Iterator<U> when running on an RDD of type T.
`mapPartitionsWithIndex(func)`	Similar to mapPartitions, but also provides func with an integer value representing the index of the partition, so func must be of type (Int, Iterator<T>) => Iterator<U> when running on an RDD of type T.
`sample(withReplacement, fraction, seed)`	Sample a fraction fraction of the data, with or without replacement, using a given random number generator seed.
`union(otherDataset)`	Return a new dataset that contains the union of the elements in the source dataset and the argument.
`intersection(otherDataset)`	Return a new RDD that contains the intersection of elements in the source dataset and the argument.

Transformations

Transformation	Description
`distinct([numTasks]))`	Return a new dataset that contains the distinct elements of the source dataset.
`groupByKey([numTasks])`	When called on a dataset of (K, V) pairs, returns a dataset of (K, Iterable<V>) pairs.
`reduceByKey(func, [numTasks])`	When called on a dataset of (K, V) pairs, returns a dataset of (K, V) pairs where the values for each key are aggregated using the given reduce function func, which must be of type (V,V) => V.
`aggregateByKey(zeroValue)(seqOp, combOp, [numTasks])`	When called on a dataset of (K, V) pairs, returns a dataset of (K, U) pairs where the values for each key are aggregated using the given combine functions and a neutral "zero" value.
`sortByKey([ascending], [numTasks])`	When called on a dataset of (K, V) pairs where K implements Ordered, returns a dataset of (K, V) pairs sorted by keys in ascending or descending order.

Transformations

Transformation	Description
`join(otherDataset, [numTasks])`	When called on datasets of type (K, V) and (K, W), returns a dataset of (K, (V, W)) pairs with all pairs of elements for each key. Outer joins are supported through leftOuterJoin,rightOuterJoin, and fullOuterJoin.
`cogroup(otherDataset, [numTasks])`	When called on datasets of type (K, V) and (K, W), returns a dataset of (K, Iterable<V>, Iterable<W>) tuples. This operation is also called groupWith.
`cartesian(otherDataset)`	When called on datasets of types T and U, returns a dataset of (T, U) pairs (all pairs of elements).
`pipe(command, [envVars])`	Pipe each partition of the RDD through a shell command, e.g. a Perl or bash script. RDD elements are written to the process's stdin and lines output to its stdout are returned as an RDD of strings.

Transformations

Transformation	Description
`coalesce(numPartitions)`	Decrease the number of partitions in the RDD to numPartitions. Useful for running operations more efficiently after filtering down a large dataset.
`repartition(numPartitions)`	Reshuffle the data in the RDD randomly to create either more or fewer partitions and balance it across them. This always shuffles all data over the network.
`repartitionAndSortWithinPartitions(partitioner)`	Repartition the RDD according to the given partitioner and, within each resulting partition, sort records by their keys. This is more efficient than calling repartition and then sorting within each partition because it can push the sorting down into the shuffle machinery.

Lab: San Francisco Parking

Spark

Map vs. Flatmap

What is the difference between Map and FlatMap?

```
lines = sc.textFile('fixtures/poem.txt')
lines.map(lambda x: x.split(' ')).collect()
lines.flatMap(lambda x: x.split(' ')).collect()
```

Note the use of closures with the **lambda** keyword.

Map vs. Flatmap

```
val rdd = sc.parallelize(Seq("Roses are red", "Violets are blue"))  // lines

rdd.collect
    res0: Array[String] = Array("Roses are red", "Violets are blue")


rdd.map(_.length).collect
    res1: Array[Int] = Array(13, 16)



rdd.flatMap(_.split(" ")).collect
    res1: Array[String] = Array("Roses", "are", "red", "Violets", "are",
"blue")
```

Actions

- Kick off evaluations and begin computation.
- Specify the result of an operation or aggregation.

Action	Description
`reduce(func)`	Aggregate the elements of the dataset using a function func (which takes two arguments and returns one). The function should be commutative and associative so that it can be computed correctly in parallel.
`collect()`	Return all the elements of the dataset as an array at the driver program. This is usually useful after a filter or other operation that returns a sufficiently small subset of the data.
`count()`	Return the number of elements in the dataset.

Actions

Action	Description
`first()`	Return the first element of the dataset (similar to take(1)).
`take(n)`	Return an array with the first n elements of the dataset. Note that this is currently not executed in parallel. Instead, the driver program computes all the elements.
`takeSample(withReplacement,num, [seed])`	Return an array with a random sample of num elements of the dataset, with or without replacement.
`takeOrdered(n, [ordering])`	Return the first n elements of the RDD using either their natural order or a custom comparator.
`countByKey()`	Only available on RDDs of type (K, V). Returns a hashmap of (K, Int) pairs with the count of each key.
`foreach(func)`	Run a function func on each element of the dataset.

Actions

Action	Description
`saveAsTextFile(path)`	Write the elements of the dataset as a text file (or set of text files) in a given directory in the local filesystem, HDFS or any other Hadoop-supported file system. Spark will call toString on each element to convert it to a line of text in the file.
`saveAsSequenceFile(path)` (Java and Scala)	Write the elements of the dataset as a Hadoop SequenceFile in a given path in the local filesystem, HDFS or any other Hadoop-supported file system. This is available on RDDs of key-value pairs that either implement Hadoop's Writable interface. In Scala, it is also available on types that are implicitly convertible to Writable (Spark includes conversions for basic types like Int, Double, String, etc).
`saveAsObjectFile(path)` (Java and Scala)	Write the elements of the dataset in a simple format using Java serialization, which can then be loaded using SparkContext.objectFile().

Persistence

Spark will persist or cache RDD slices in memory on each node during operations.

Fault tolerant - in case of failure, Spark can rebuild the RDD from the lineage, automatically recreating the slice.

Super fast - will allow multiple operations on the same data set.

You can mark an RDD to be persisted with the cache method on an RDD along with a storage level.

Python objects are always pickles.

Lab: Movie Ratings

Spark