This Lecture

Programming Spark

Resilient Distributed Datasets (RDDs)

Creating an RDD

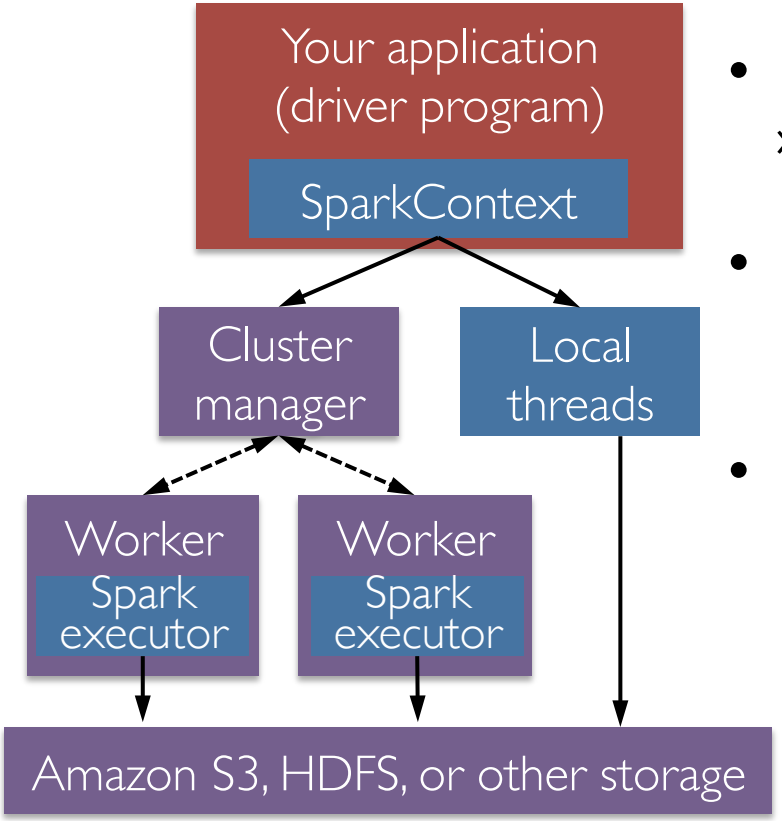
Spark Transformations and Actions

Spark Programming Model

Python Spark (pySpark)

- We are using the Python programming interface to Spark (pySpark)
- pySpark provides an easy-to-use programming abstraction and parallel runtime:
  - » "Here's an operation, run it on all of the data"
- RDDs are the key concept

Spark Driver and Workers



- A Spark program is two programs:
  - » A driver program and a workers program
- Worker programs run on cluster nodes or in local threads
- RDDs are distributed across workers

Spark Context

- A Spark program first creates a **`SparkContext`** object
  - » Tells Spark how and where to access a cluster
  - » pySpark shell and Databricks Cloud automatically create the **`sc`** variable
  - » iPython and programs must use a constructor to create a new **`SparkContext`**
- Use **`SparkContext`** to create RDDs

In the labs, we create the SparkContext for you

Spark Essentials: Master

- The **`master`** parameter for a **`SparkContext`** determines which type and size of cluster to use

Master Parameter	Description
`local`	run Spark locally with one worker thread (no parallelism)
`local[K]`	run Spark locally with K worker threads (ideally set to number of cores)
`spark://HOST:PORT`	connect to a Spark standalone cluster; PORT depends on config (7077 by default)
`mesos://HOST:PORT`	connect to a Mesos cluster; PORT depends on config (5050 by default)

In the labs, we set the master parameter for you

Resilient Distributed Datasets

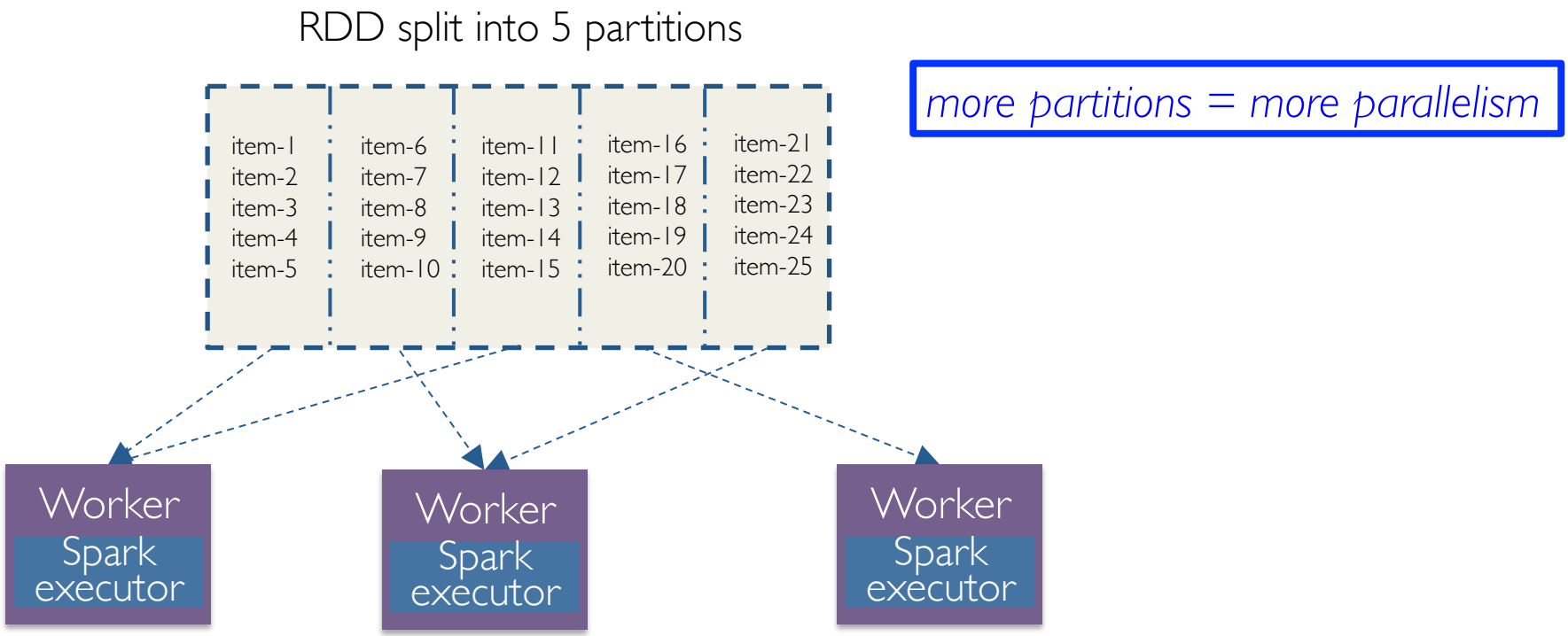
- The primary abstraction in Spark
  - » Immutable once constructed
  - » Track lineage information to efficiently recompute lost data
  - » Enable operations on collection of elements in parallel

- You construct RDDs
  - » by *parallelizing* existing Python collections (lists)
  - » by *transforming* an existing RDDs
  - » from *files* in HDFS or any other storage system

RDDs

- Programmer specifies number of partitions for an RDD

(Default value used if unspecified)

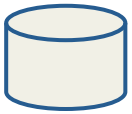


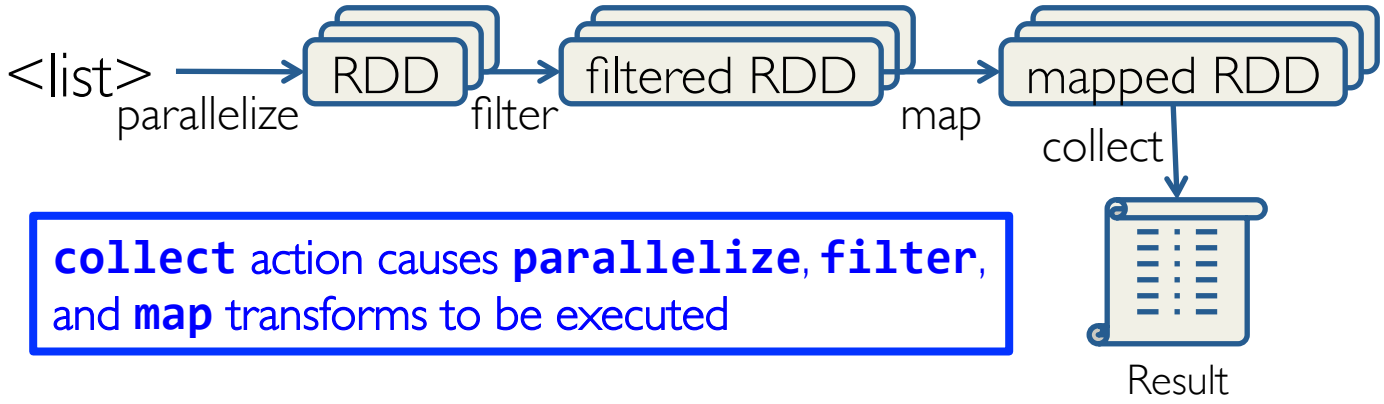
more partitions = more parallelism

RDDs

- Two types of operations: *transformations* and *actions*
- Transformations are lazy (*not computed immediately*)
- Transformed RDD is executed when action runs on it
- Persist (cache) RDDs in memory or disk

Working with RDDs

- Create an RDD from a data source:  <list>
- Apply transformations to an RDD: map filter
- Apply actions to an RDD: collect count



`collect` action causes **`parallelize`**, **`filter`**, and **`map`** transforms to be executed

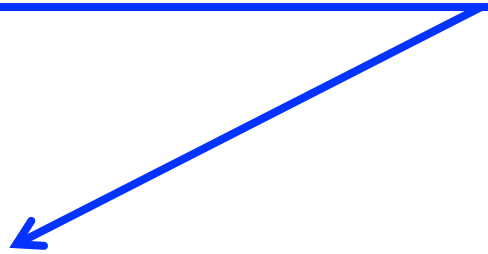
Creating an RDD

- Create RDDs from Python collections (lists)

```
>>> data = [1, 2, 3, 4, 5]
>>> data
[1, 2, 3, 4, 5]


>>> rDD = sc.parallelize(data, 4)
>>> rDD
ParallelCollectionRDD[0] at parallelize at PythonRDD.scala:229
```

No computation occurs with **`sc.parallelize()`**

- Spark only records how to create the RDD with four partitions



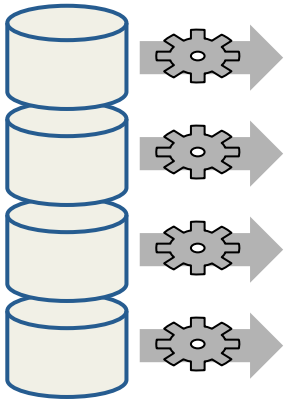
Creating RDDs

- From HDFS, text files, Hypertable, Amazon S3, Apache Hbase, SequenceFiles, any other Hadoop **`InputFormat`**, and directory or glob wildcard: **`/data/201404*`**

```
>>> distFile = sc.textFile("README.md", 4)

>>> distFile

MappedRDD[2] at textFile at

   NativeMethodAccessorImpl.java:-2
```

Creating an RDD from a File

```
distFile = sc.textFile("...", 4)
```



- RDD distributed in 4 partitions
- Elements are lines of input
- *Lazy evaluation* means no execution happens now

Some Transformations

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
`distinct([`*`numTasks`*`]))`	return a new dataset that contains the distinct elements of the source dataset
`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)

Review: Python `lambda` Functions

- Small anonymous functions (not bound to a name)
  `lambda a, b: a + b`
  » returns the sum of its two arguments
- Can use lambda functions wherever function objects are required
- Restricted to a single expression

Transformations

```
>>> rdd = sc.parallelize([1, 2, 3, 4])
>>> rdd.map(lambda x: x * 2)
RDD: [1, 2, 3, 4] → [2, 4, 6, 8]

>>> rdd.filter(lambda x: x % 2 == 0)
RDD: [1, 2, 3, 4] → [2, 4]

>>> rdd2 = sc.parallelize([1, 4, 2, 2, 3])
>>> rdd2.distinct()
RDD: [1, 4, 2, 2, 3] → [1, 4, 2, 3]
```

Function literals (green) are closures automatically passed to workers

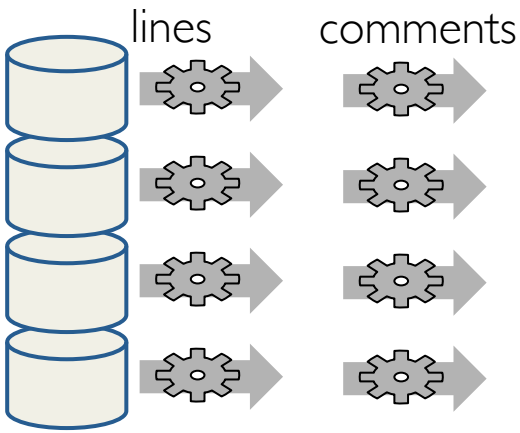
Transformations

```
>>> rdd = sc.parallelize([1, 2, 3])
>>> rdd.Map(lambda x: [x, x+5])
RDD: [1, 2, 3] → [[1, 6], [2, 7], [3, 8]]

>>> rdd.flatMap(lambda x: [x, x+5])
RDD: [1, 2, 3] → [1, 6, 2, 7, 3, 8]
```

Function literals (green) are closures automatically passed to workers

Transforming an RDD

```
lines = sc.textFile("...", 4)

comments = lines.filter(isComment)
```



Lazy evaluation means nothing executes – Spark saves recipe for transforming source

Spark Actions

- Cause Spark to execute recipe to transform source
- Mechanism for getting results out of Spark

Some Actions

Action	Description
`reduce(`*`func`*`)`	aggregate dataset's elements using function *func*. *func* takes two arguments and returns one, and is commutative and associative so that it can be computed correctly in parallel
`take(`*`n`*`)`	return an array with the first *n* elements
`collect()`	return all the elements as an array WARNING: make sure will fit in driver program
`takeOrdered(`*`n, key=func`*`)`	return n elements ordered in ascending order or as specified by the optional key function

Getting Data Out of RDDs

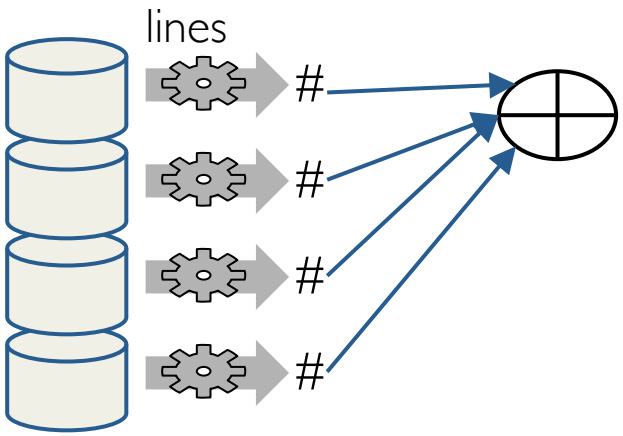
```
>>> rdd = sc.parallelize([1, 2, 3])
>>> rdd.reduce(lambda a, b: a * b)
Value: 6

>>> rdd.take(2)
Value: [1,2] # as list

>>> rdd.collect()
Value: [1,2,3] # as list
```

Getting Data Out of RDDs

```
>>> rdd = sc.parallelize([5,3,1,2])
>>> rdd.takeOrdered(3, lambda s: -1 * s)
Value: [5,3,2] # as list
```

Spark Programming Model

```
lines = sc.textFile("...", 4)

print lines.count()
```

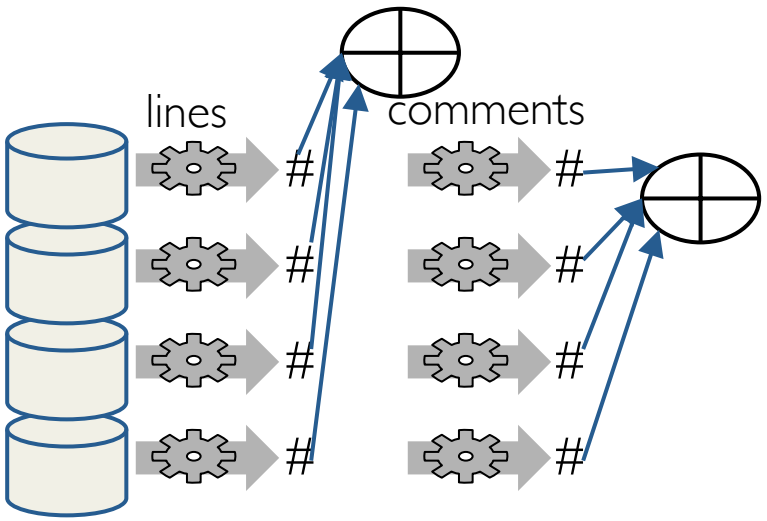


`count()` causes Spark to:

- read data
- sum within partitions
- combine sums in driver

Spark Programming Model

```
lines = sc.textFile("...", 4)
comments = lines.filter(isComment)
print lines.count(), comments.count()
```

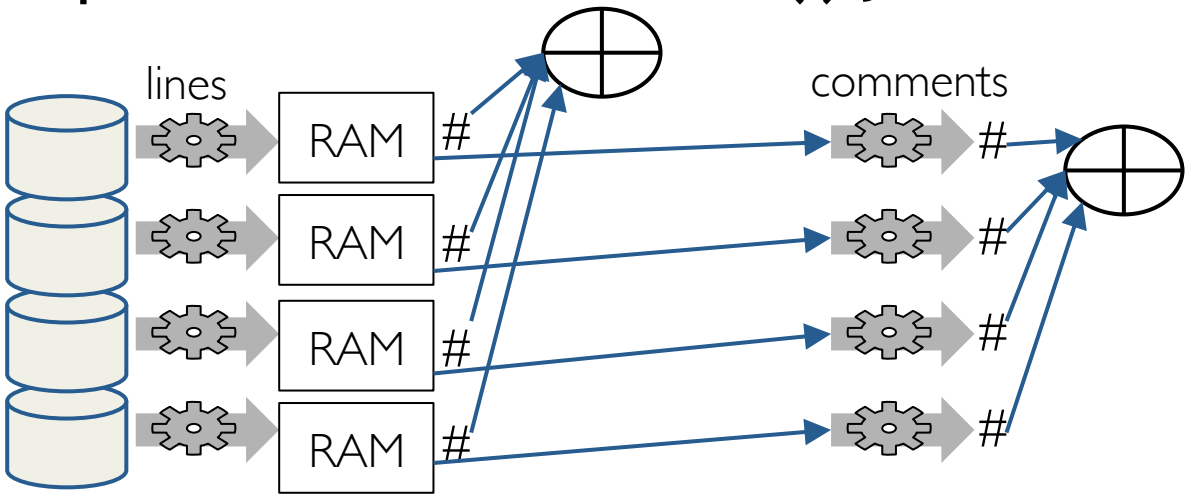


Spark recomputes **lines**:

- read data (again)
- sum within partitions
- combine sums in driver

Caching RDDs

```
lines = sc.textFile("...", 4)
lines.cache() # save, don't recompute!
comments = lines.filter(isComment)
print lines.count(),comments.count()
```



Spark Program Lifecycle

1. Create RDDs from external data or parallelize a collection in your driver program
2. Lazily transform them into new RDDs
3. **`cache()`** some RDDs for reuse
4. Perform actions to execute parallel computation and produce results

Spark Key-Value RDDs

- Similar to Map Reduce, Spark supports Key-Value pairs
- Each element of a Pair RDD is a pair tuple

```
>>> rdd = sc.parallelize([(1, 2), (3, 4)])
RDD: [(1, 2), (3, 4)]
```

Some Key-Value Transformations

Key-Value Transformation	Description
`reduceByKey(`*`func`*`)`	return a new distributed 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
`sortByKey()`	return a new dataset (K,V) pairs sorted by keys in ascending order
`groupByKey()`	return a new dataset of (K, Iterable<V>) pairs

Key-Value Transformations

```
>>> rdd = sc.parallelize([(1,2), (3,4), (3,6)])
>>> rdd.reduceByKey(lambda a, b: a + b)
RDD: [(1,2), (3,4), (3,6)] → [(1,2), (3,10)]

>>> rdd2 = sc.parallelize([(1,'a'), (2,'c'), (1,'b')])
>>> rdd2.sortByKey()
RDD: [(1,'a'), (2,'c'), (1,'b')] →
                [(1,'a'), (1,'b'), (2,'c')]
```

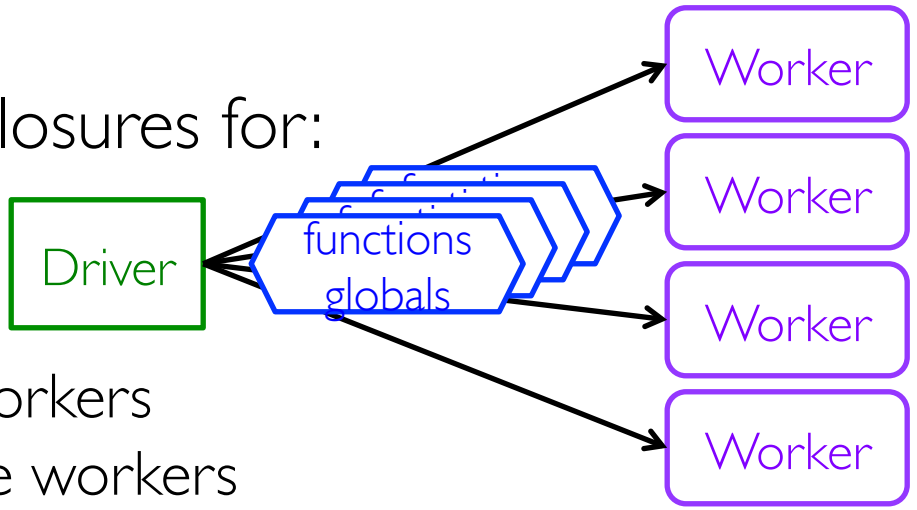
Key-Value Transformations

```
>>> rdd2 = sc.parallelize([(1,'a'), (2,'c'), (1,'b')])
>>> rdd2.groupByKey()
RDD: [(1,'a'), (1,'b'), (2,'c')] →
                [(1,['a','b']), (2,['c'])]
```

Be careful using **`groupByKey()`** as it can cause a lot of data movement across the network and create large Iterables at workers

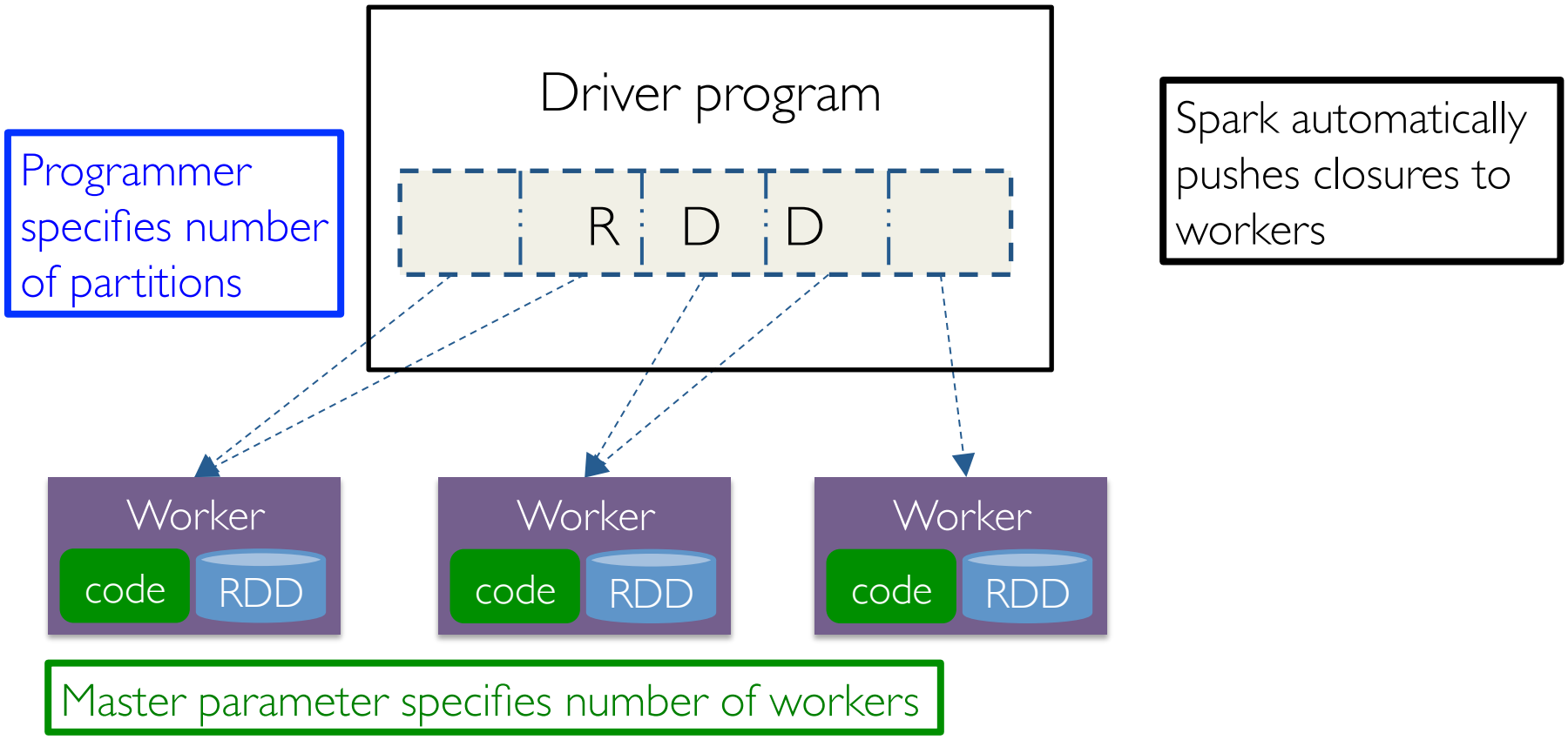
pySpark Closures

- Spark automatically creates closures for:
  - » Functions that run on RDDs at workers
  - » Any global variables used by those workers



- One closure per worker
  - » Sent for ***every*** task
  - » No communication between workers
  - » Changes to global variables at workers are not sent to driver

Summary


Driver program
R
D
D
Programmer specifies number of partitions
Spark automatically pushes closures to workers
Worker
code
RDD
Worker
code
RDD
Worker
code
RDD
Master parameter specifies number of workers