ÉCOLE POLYTECHNIQUE
FÉDÉRALE DE LAUSANNE

Data-Parallel to Distributed Data-Parallel

Big Data Analysis with Scala and Spark

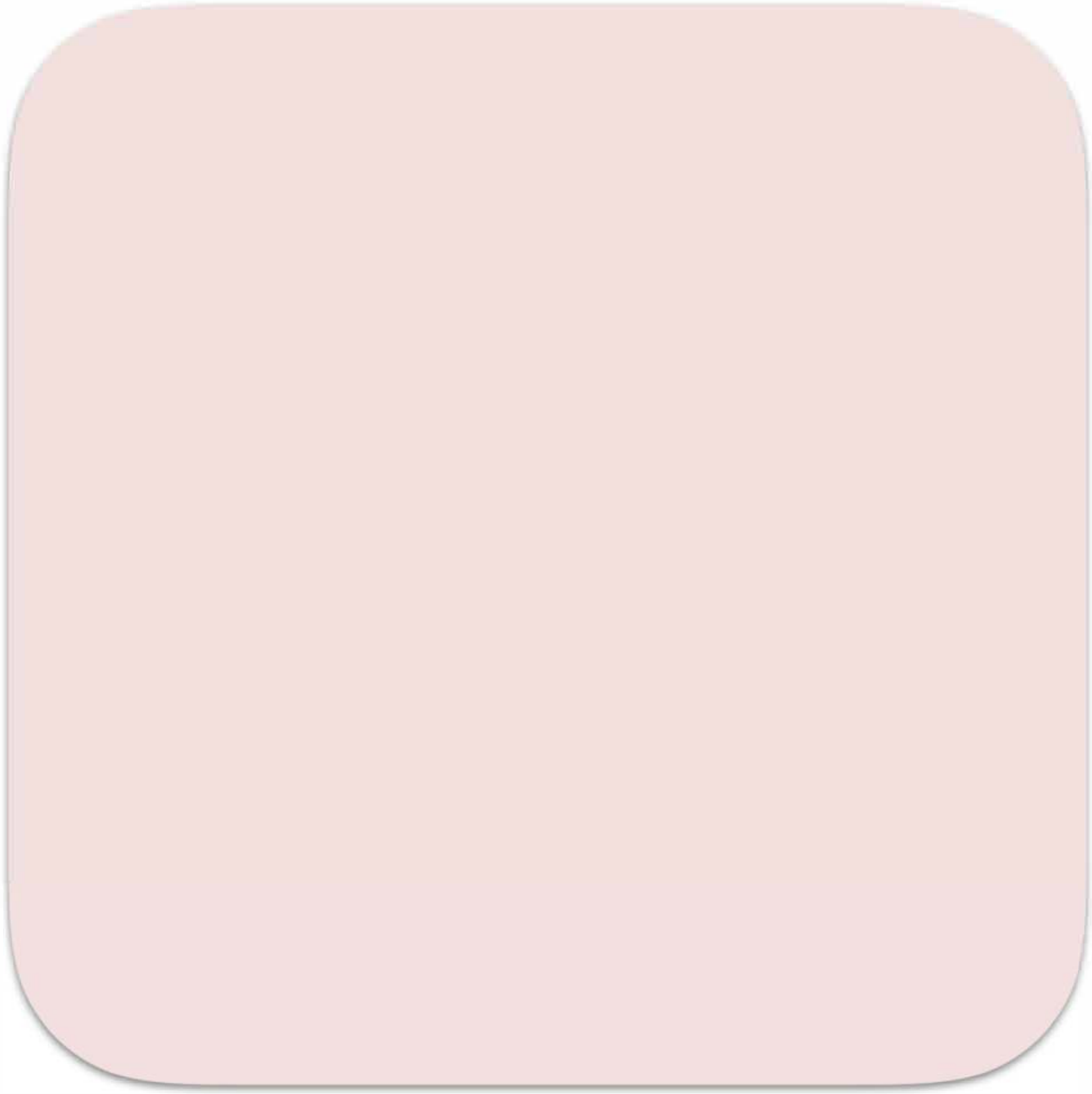
Heather Miller

Visualizing **Shared Memory** Data Parallelism

What does data-parallel look like?

Visualizing **Shared Memory** Data Parallelism

What does data-parallel look like?

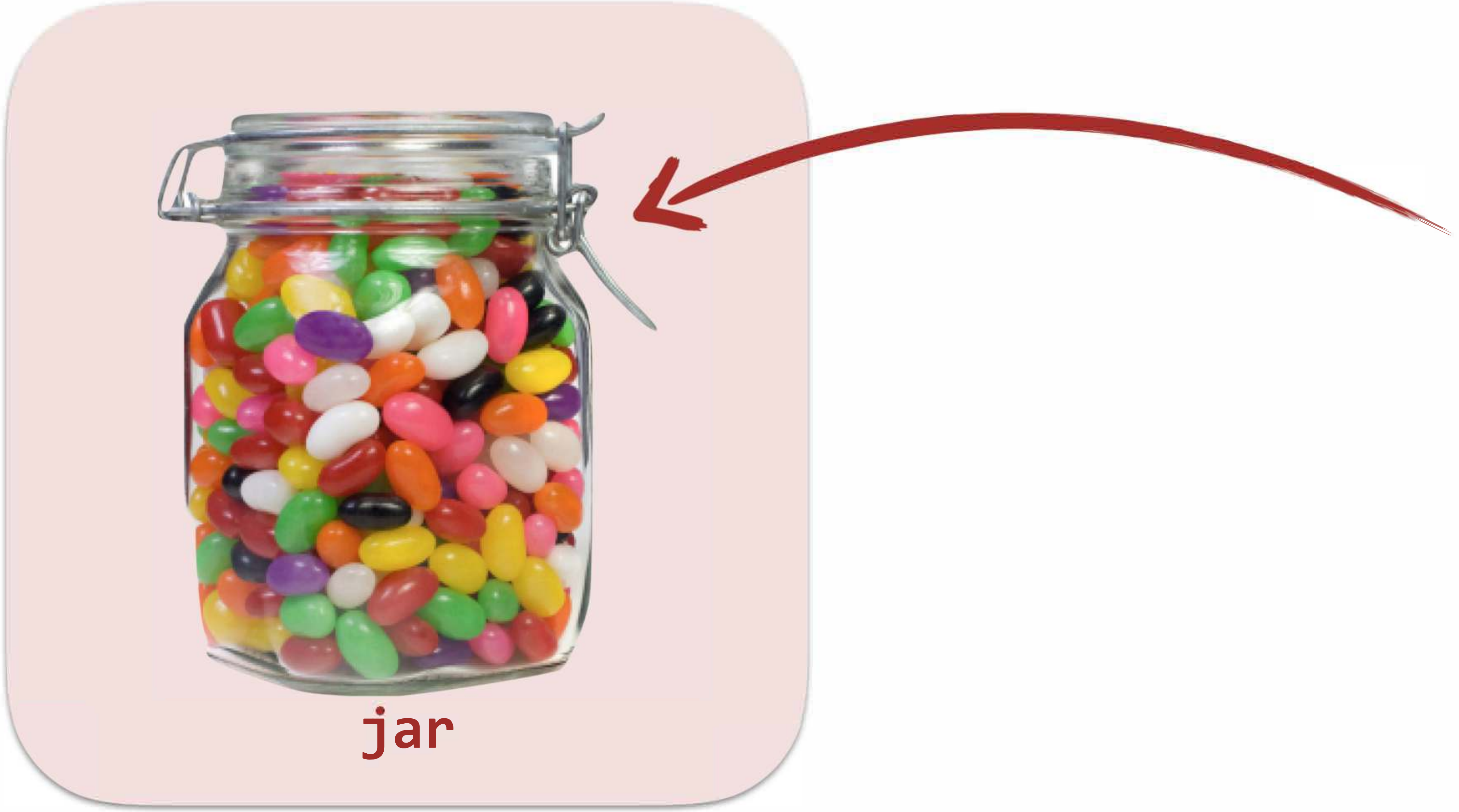


Compute Node

(Shared Memory)

Visualizing **Shared Memory** Data Parallelism

What does data-parallel look like?



Collection

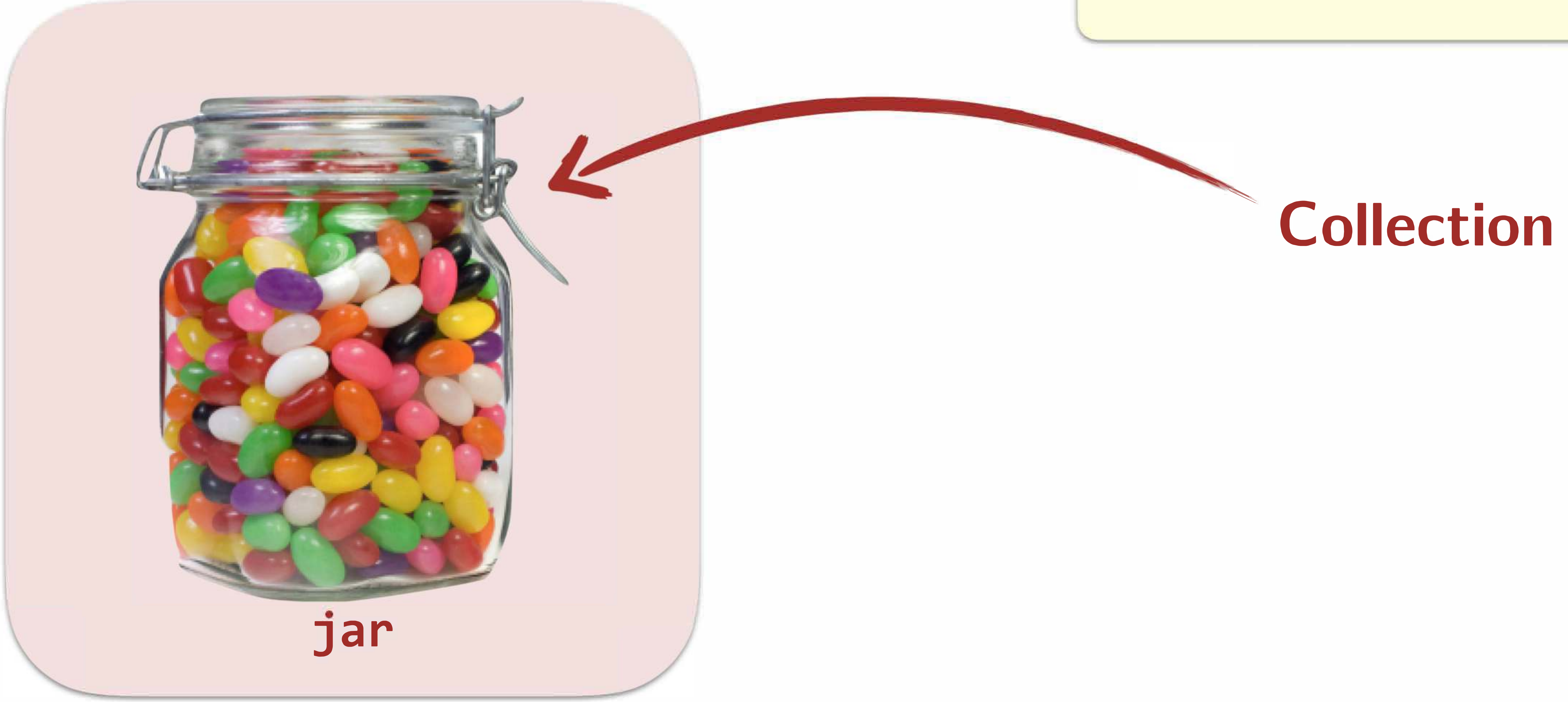
Compute Node

(Shared Memory)

Visualizing **Shared Memory** Data Parallelism

What does data-parallel look like?

```
val res =
  jar.map(jellyBean => doSomething(jellyBean))
```



Compute Node

(Shared Memory)

Visualizing **Shared Memory** Data Parallelism

What does data-parallel look like?

```
val res =
  jar.map(jellyBean => doSomething(jellyBean))
```

Compute Node

(Shared Memory)

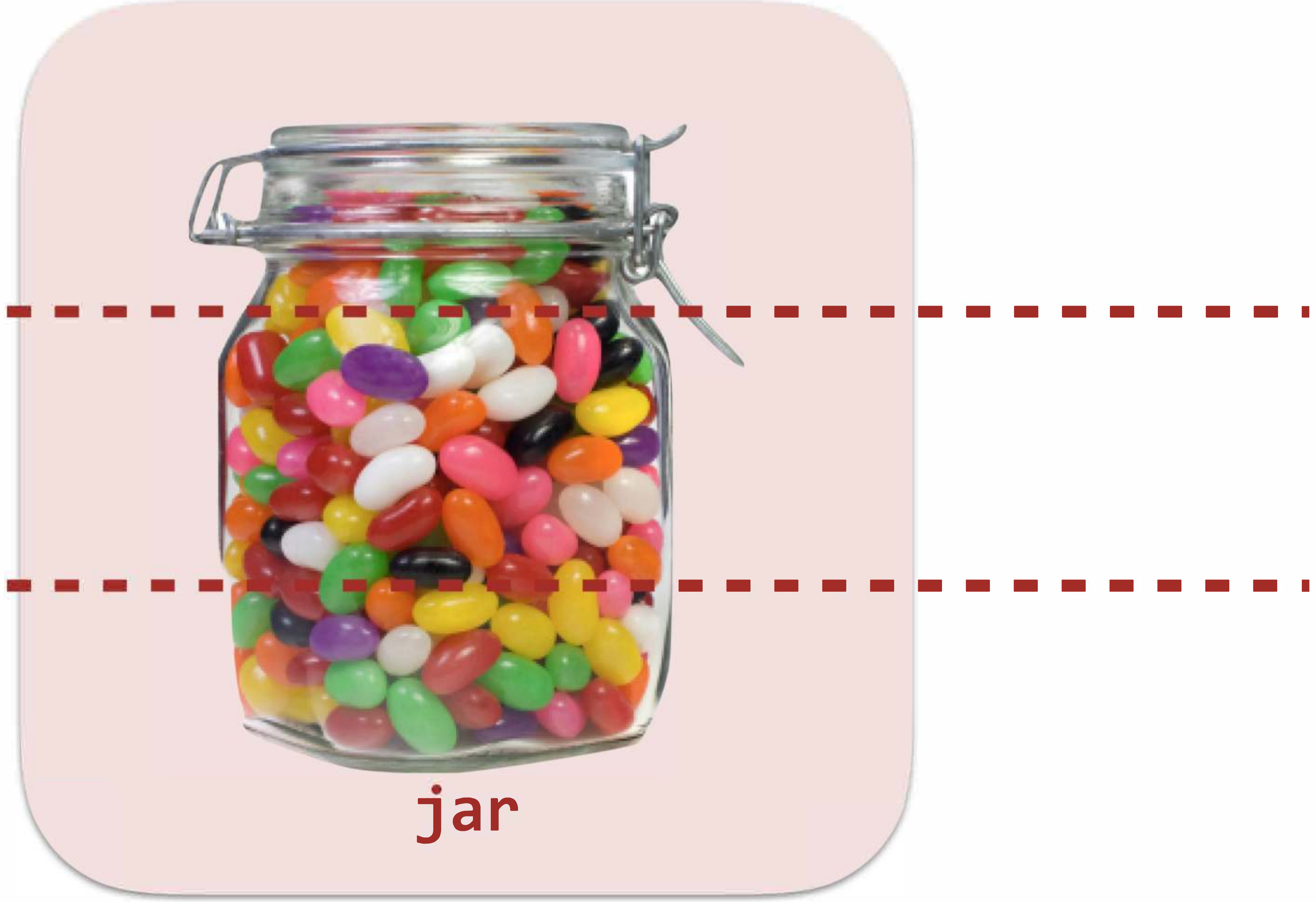
Shared memory data parallelism:

- Split the data.
- Workers/threads independently operate on the data shards in parallel.
- Combine when done (if necessary).

Visualizing **Shared Memory** Data Parallelism

What does data-parallel look like?

```
val res =
  jar.map(jellyBean => doSomething(jellyBean))
```



Compute Node

(Shared Memory)

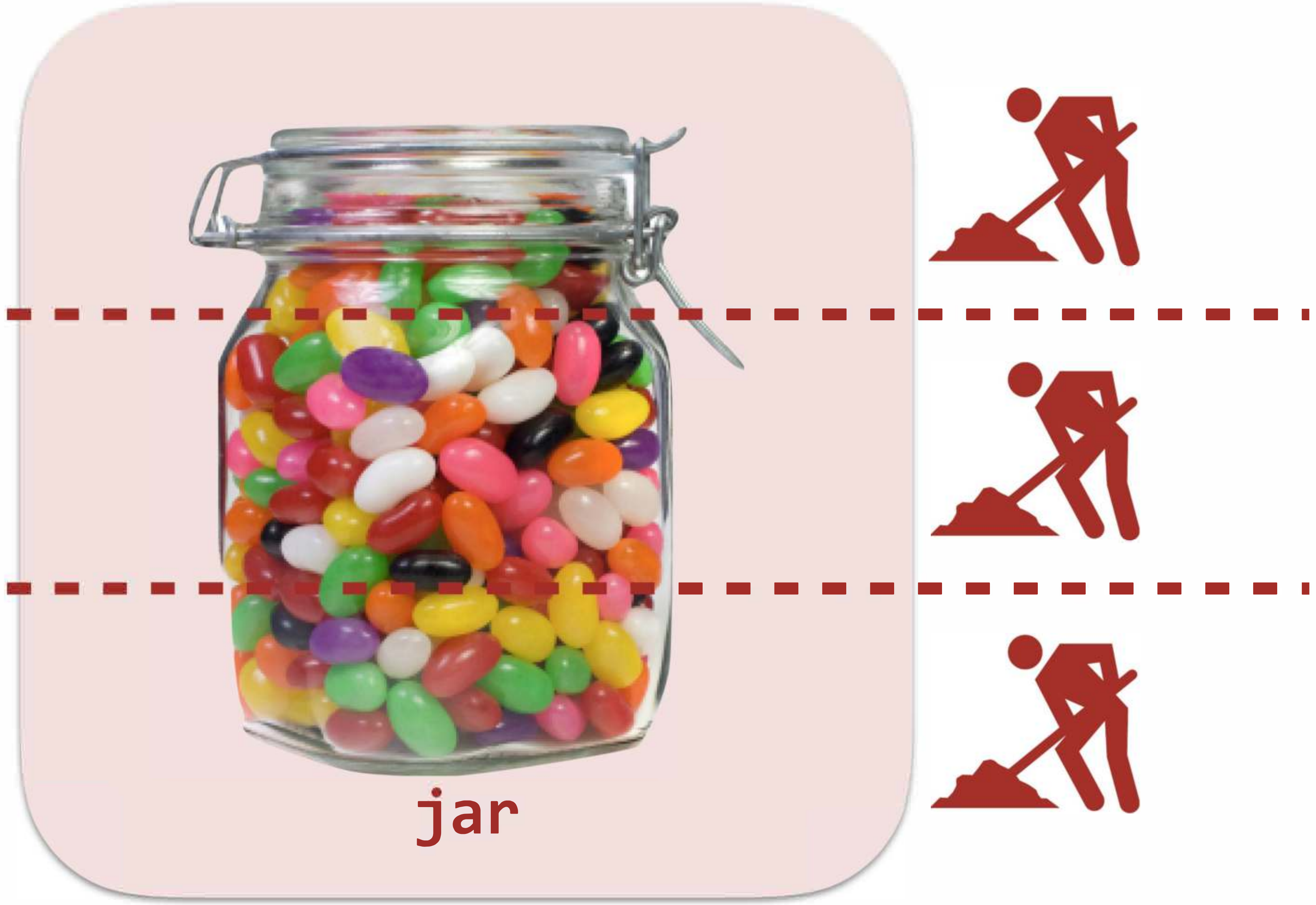
Shared memory data parallelism:

- Split the data.
- Workers/threads independently operate on the data shards in parallel.
- Combine when done (if necessary).

Visualizing Shared Memory Data Parallelism

What does data-parallel look like?

```
val res =
  jar.map(jellyBean => doSomething(jellyBean))
```



Compute Node

(Shared Memory)

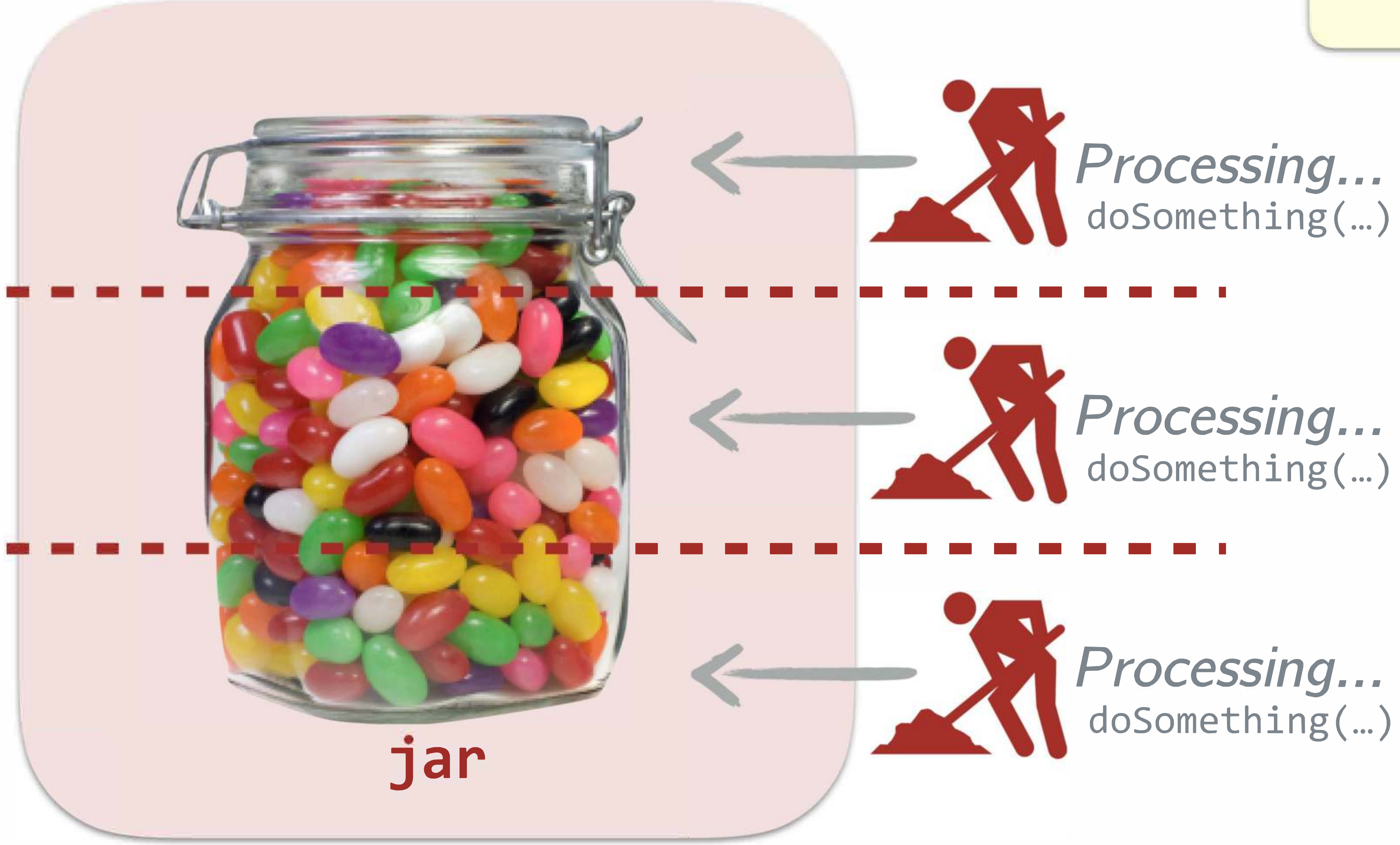
Shared memory data parallelism:

- Split the data.
- Workers/threads independently operate on the data shards in parallel.
- Combine when done (if necessary).

Visualizing **Shared Memory** Data Parallelism

What does data-parallel look like?

```
val res =
  jar.map(jellyBean => doSomething(jellyBean))
```



Compute Node

(Shared Memory)

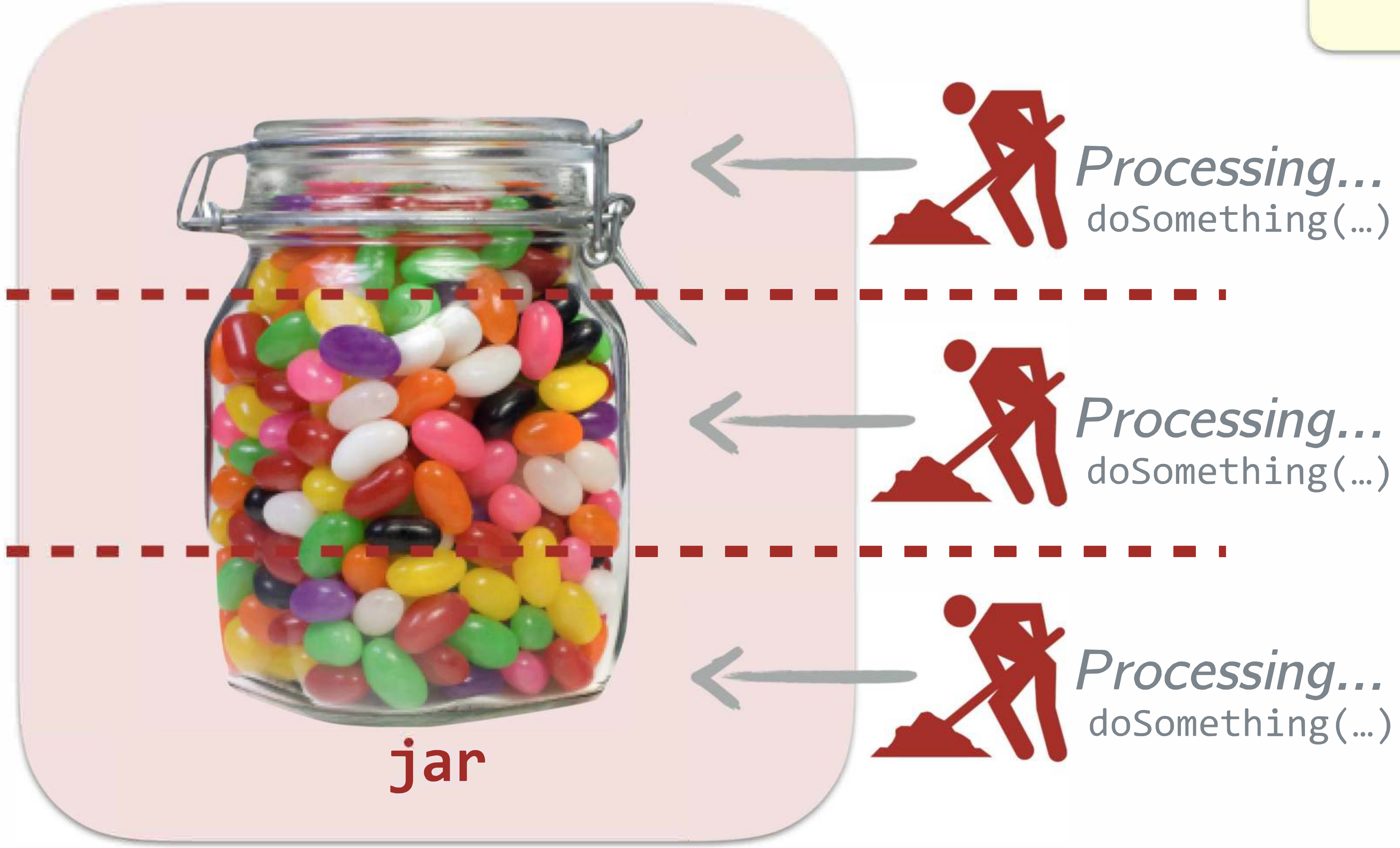
Shared memory data parallelism:

- Split the data.
- Workers/threads independently operate on the data shards in parallel.
- Combine when done (if necessary).

Visualizing Shared Memory Data Parallelism

What does data-parallel look like?

```
val res =
  jar.map(jellyBean => doSomething(jellyBean))
```



Compute Node
(Shared Memory)

Shared memory data parallelism:

- Split the data.
- Workers/threads independently operate on the data shards in parallel.
- Combine when done (if necessary).

Scala's Parallel Collections is a collections abstraction over shared memory data-parallel execution.

Visualizing **Distributed** Data-Parallelism

What does **distributed** data-parallel look like?

Shared memory data parallelism:

- Split the data.
- Workers/threads independently operate on the data shards in parallel.
- Combine when done (if necessary).

Visualizing **Distributed** Data-Parallelism

What does **distributed** data-parallel look like?

Distributed
~~Shared-memory~~ data parallelism:

- Split the data over several nodes.
- Nodes independently operate on the data shards in parallel.
- Combine when done (if necessary).

Visualizing **Distributed** Data-Parallelism

What does **distributed** data-parallel look like?

Distributed data parallelism:

- Split the data over several nodes.
- Nodes independently operate on the data shards in parallel.
- Combine when done (if necessary).

Visualizing **Distributed** Data-Parallelism

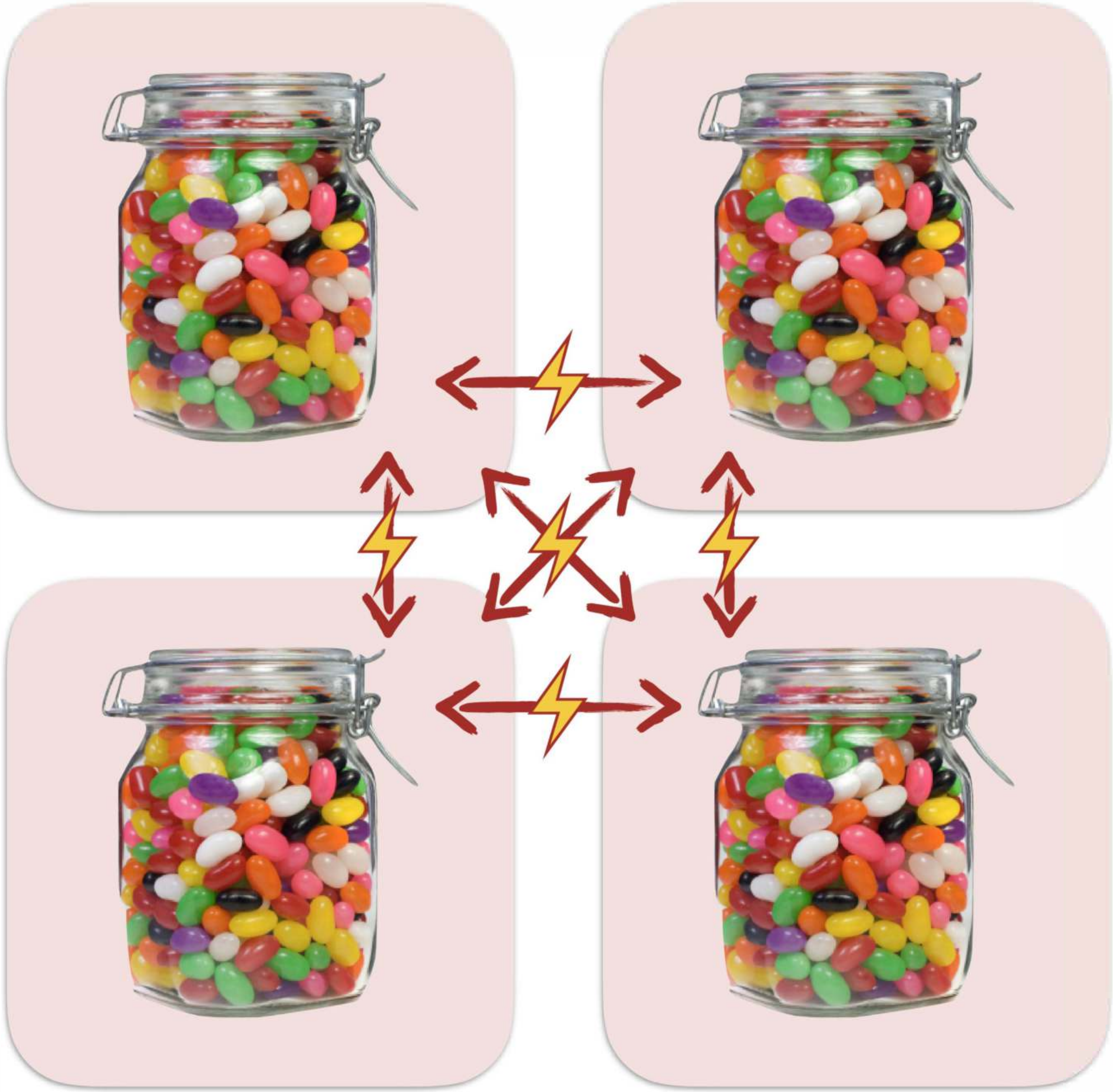
What does **distributed** data-parallel look like?

Distributed data parallelism:

- Split the data over several nodes.
- Nodes independently operate on the data shards in parallel.
- Combine when done (if necessary).

Visualizing **Distributed** Data-Parallelism

What does **distributed** data-parallel look like?



Distributed data parallelism:

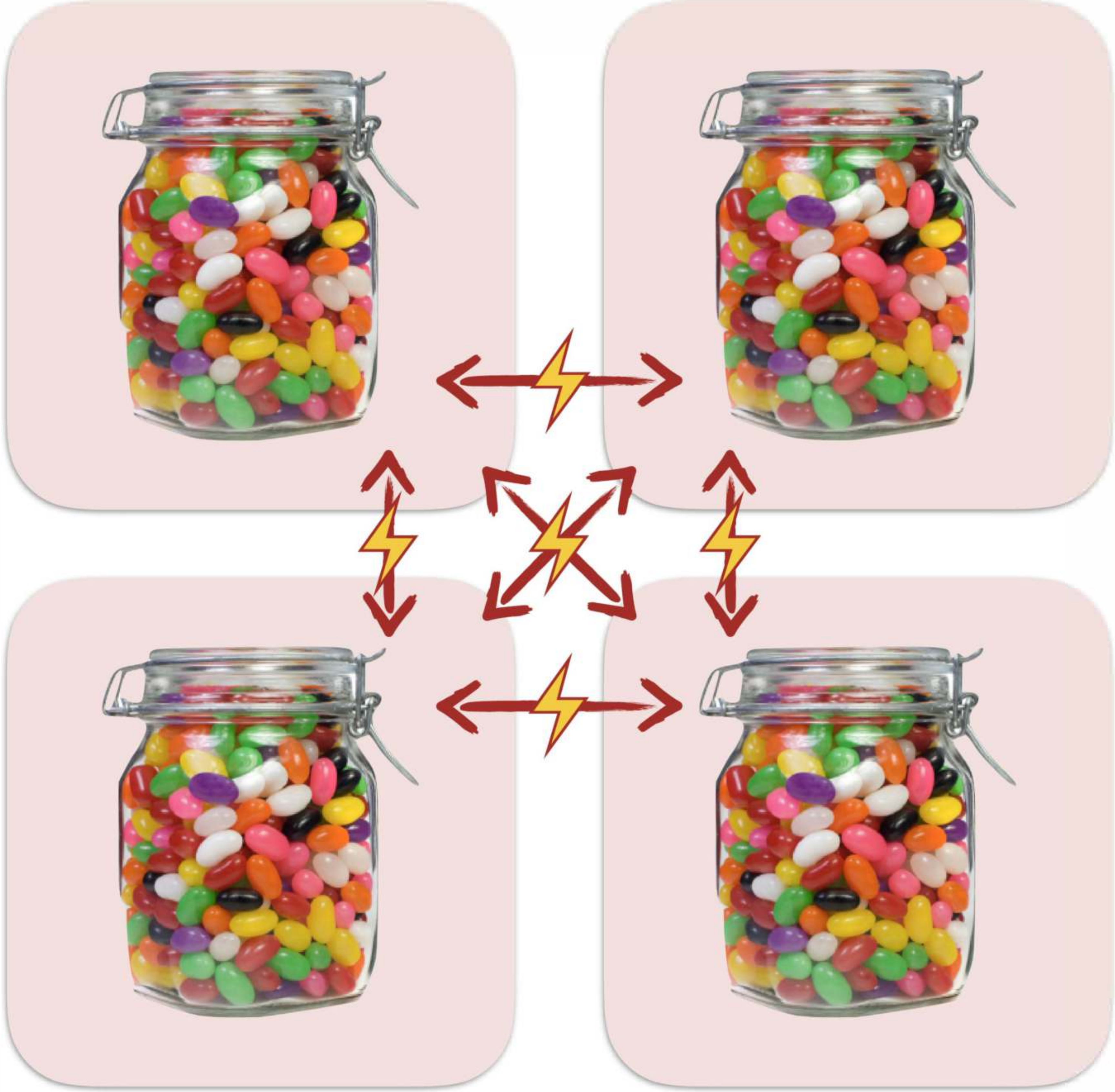
- Split the data over several nodes.
- Nodes independently operate on the data shards in parallel.
- Combine when done (if necessary).

New concern:

Now we have to worry about network latency between workers.

Visualizing **Distributed** Data-Parallelism

What does **distributed** data-parallel look like?



```
val res =
  jar.map(jellyBean => doSomething(jellyBean))
```

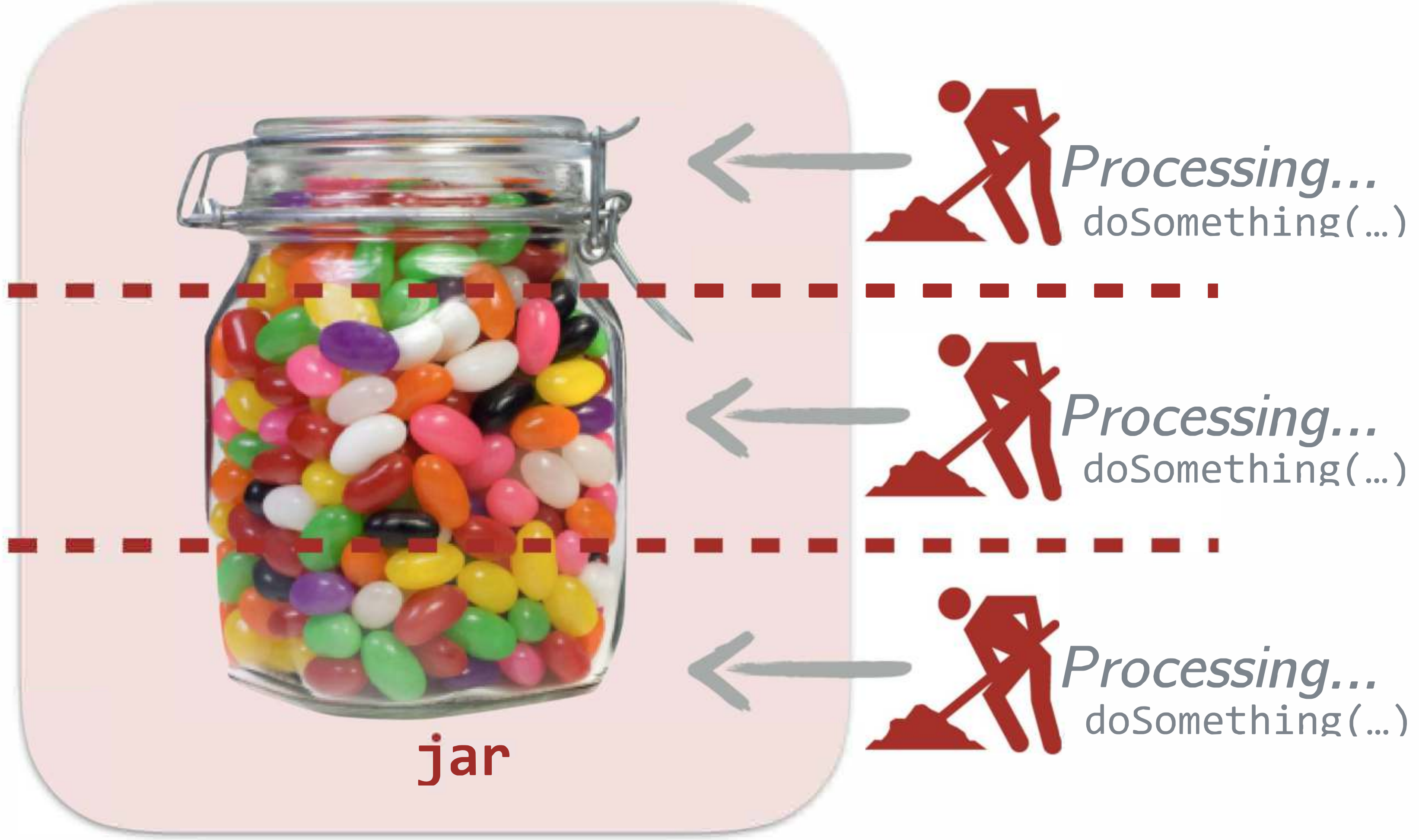
Distributed data parallelism:

- Split the data over several nodes.
- Nodes independently operate on the data shards in parallel.
- Combine when done (if necessary).

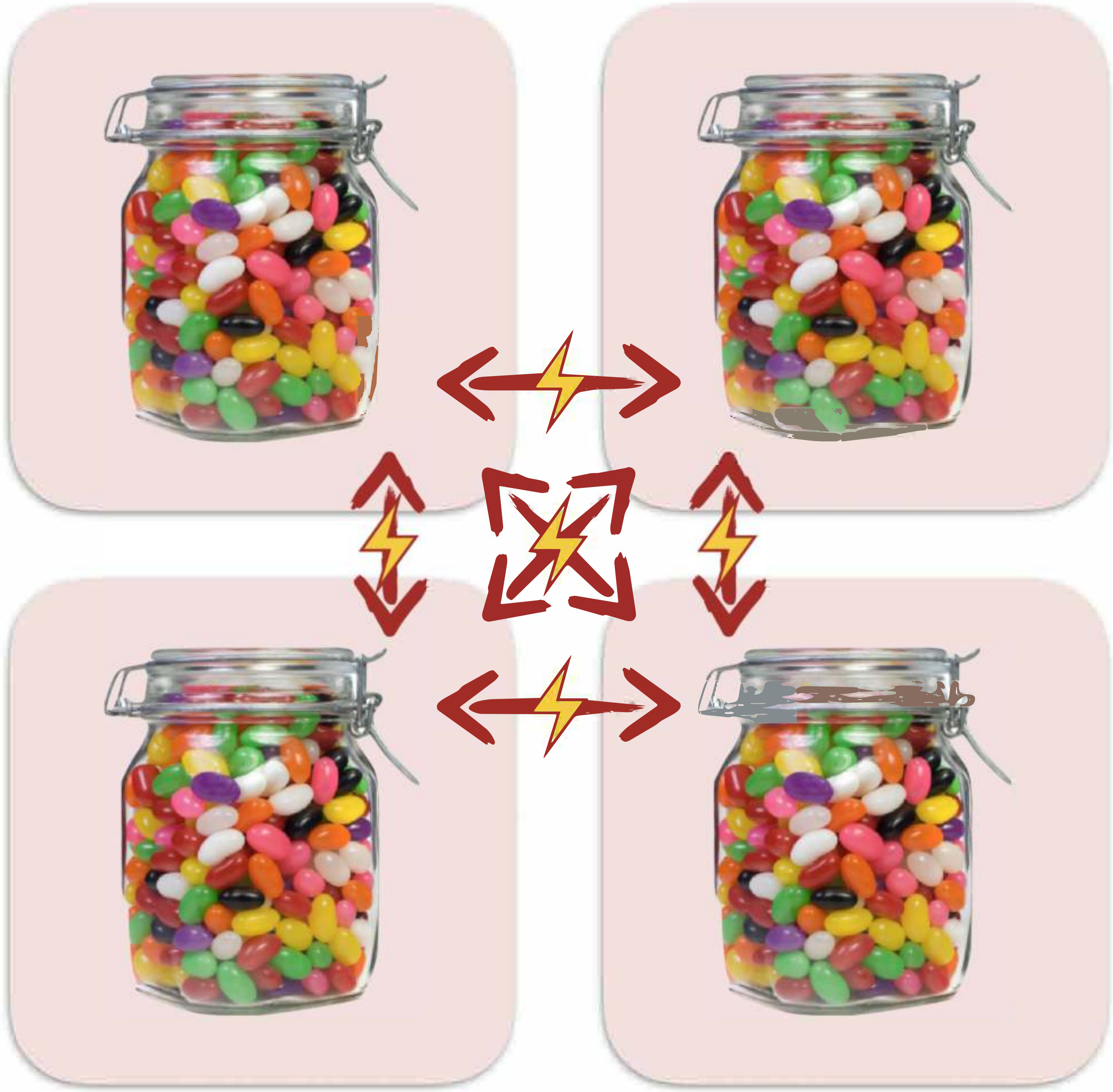
However, like parallel collections, we can keep collections abstraction over distributed data-parallel execution.

Data-Parallel to **Distributed** Data-Parallel

Shared memory:



Distributed:

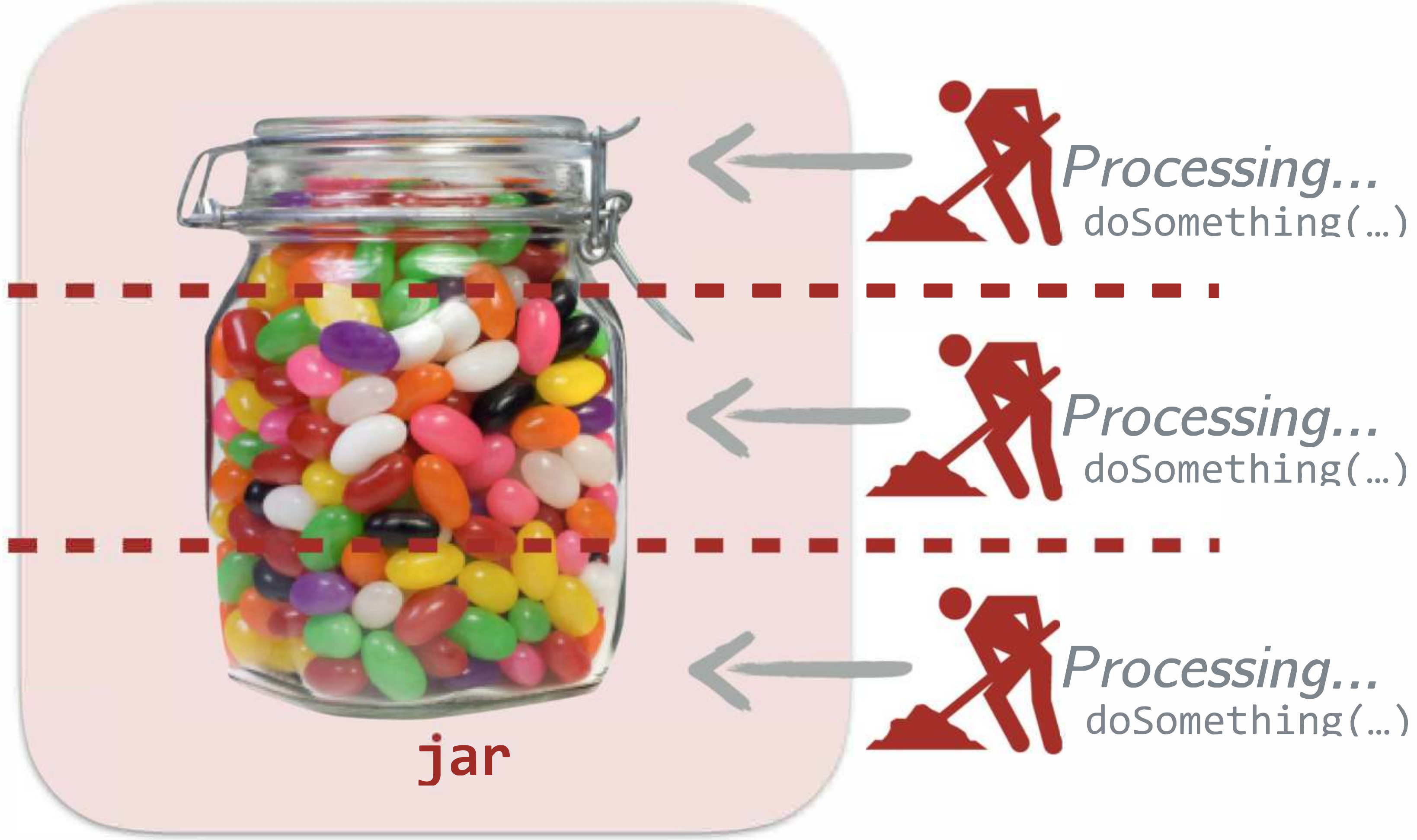


Shared memory case: Data-parallel programming model. Data partitioned in memory and operated upon in parallel.

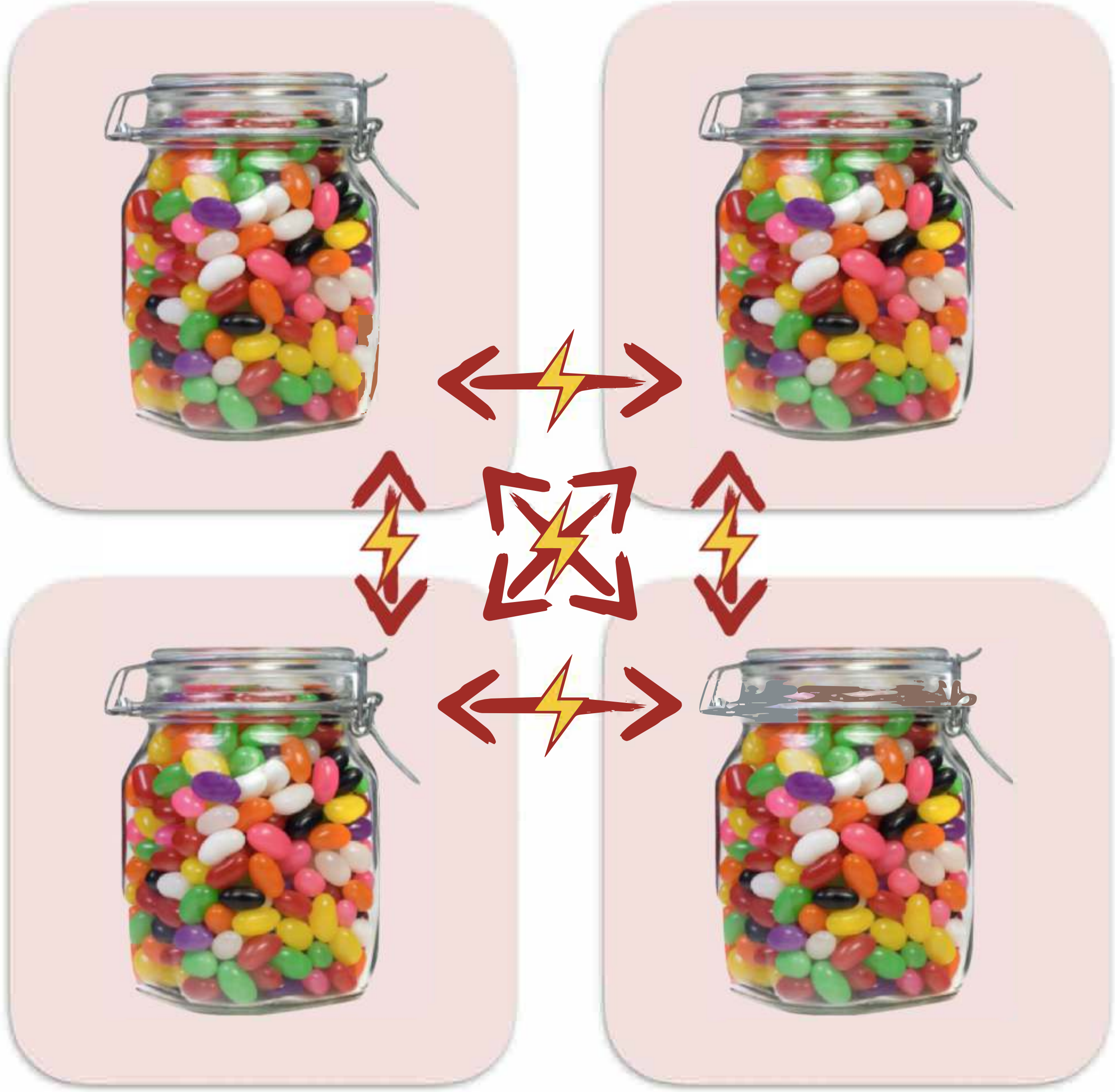
Distributed case: Data-parallel programming model. Data partitioned between machines, network in between, operated upon in parallel.

Data-Parallel to **Distributed** Data-Parallel

Shared memory:



Distributed:



Overall, most all properties we learned about related to shared memory data-parallel collections can be applied to their distributed counterparts.

E.g., watch out for non-associative reduction operations!

.reduce(_-_)

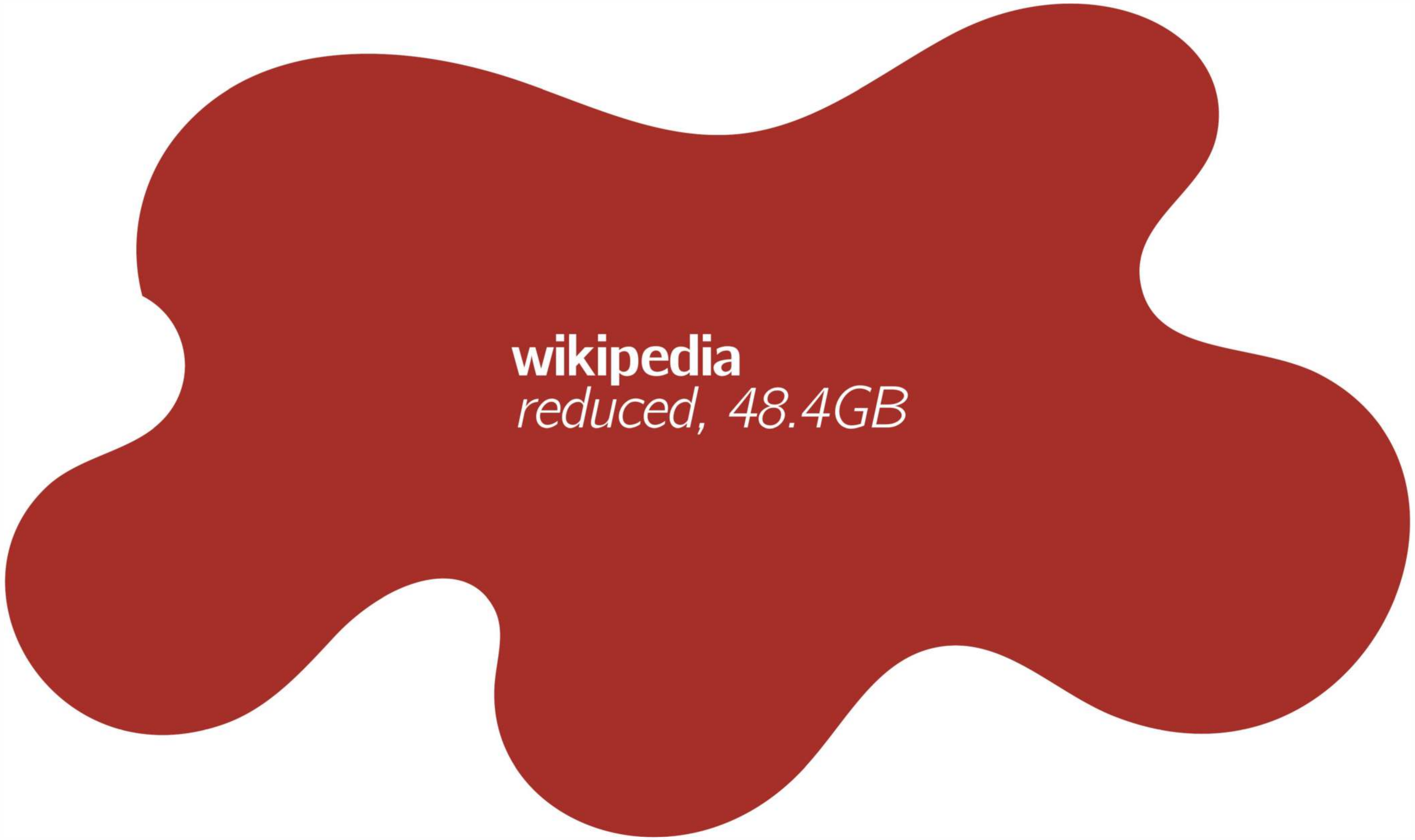
However, must now consider **latency** when using our model.

Apache Spark

Throughout this part of the course we will use the **Apache Spark** framework for distributed data-parallel programming.

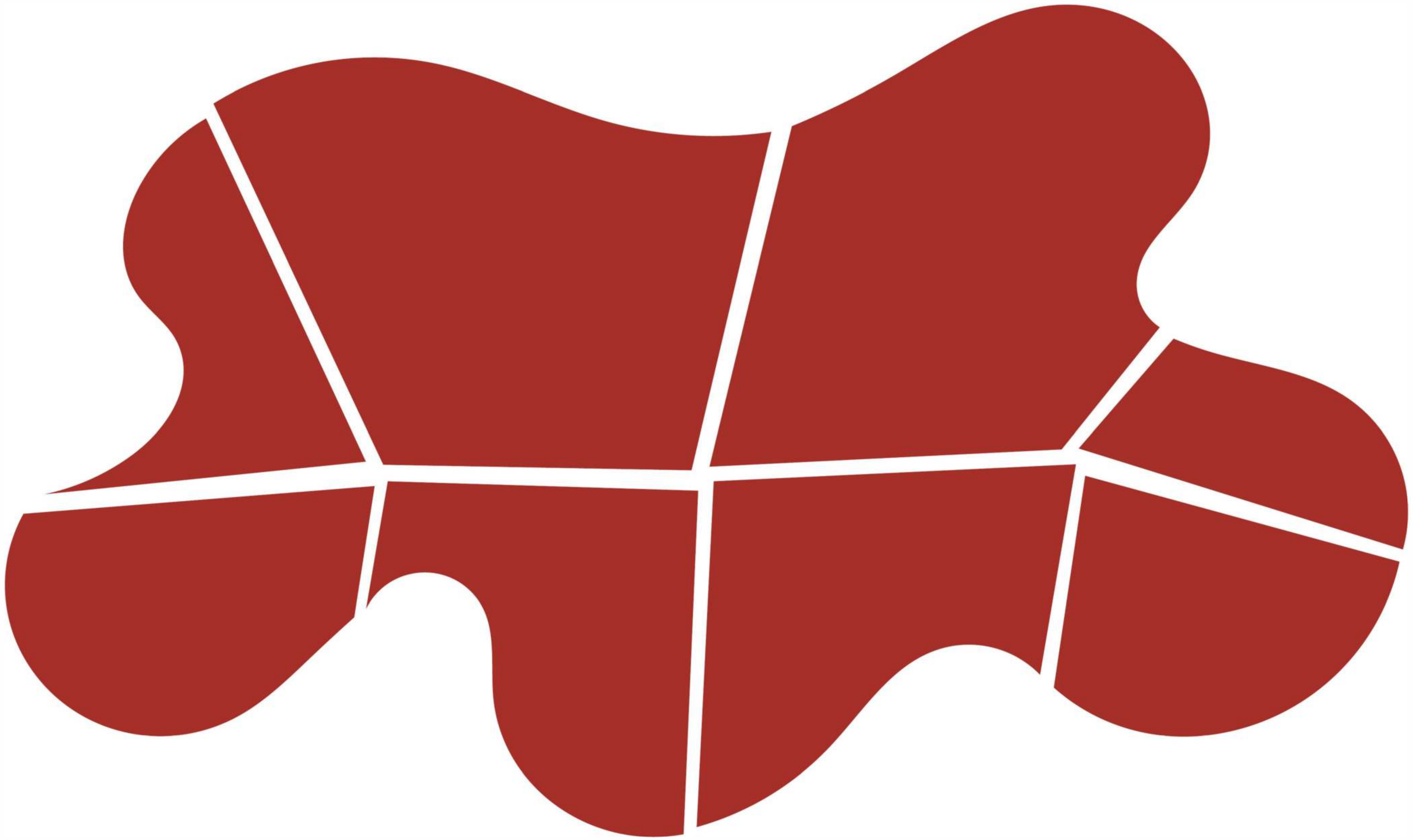
Spark implements a distributed data parallel model called **Resilient Distributed Datasets (RDDs)**

Distributed Data-Parallel: High Level Illustration



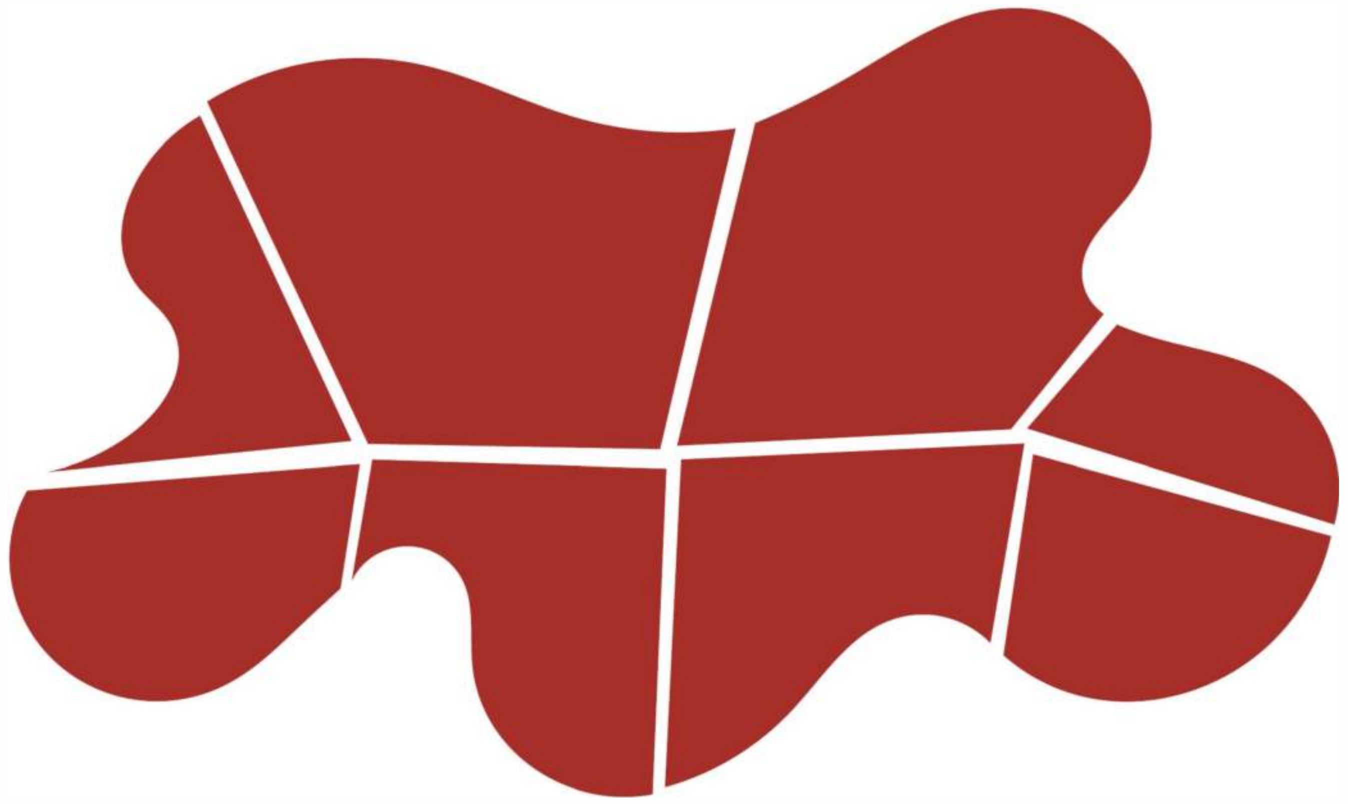
Given some large dataset that can't fit into memory on a single node...

Distributed Data-Parallel: High Level Illustration



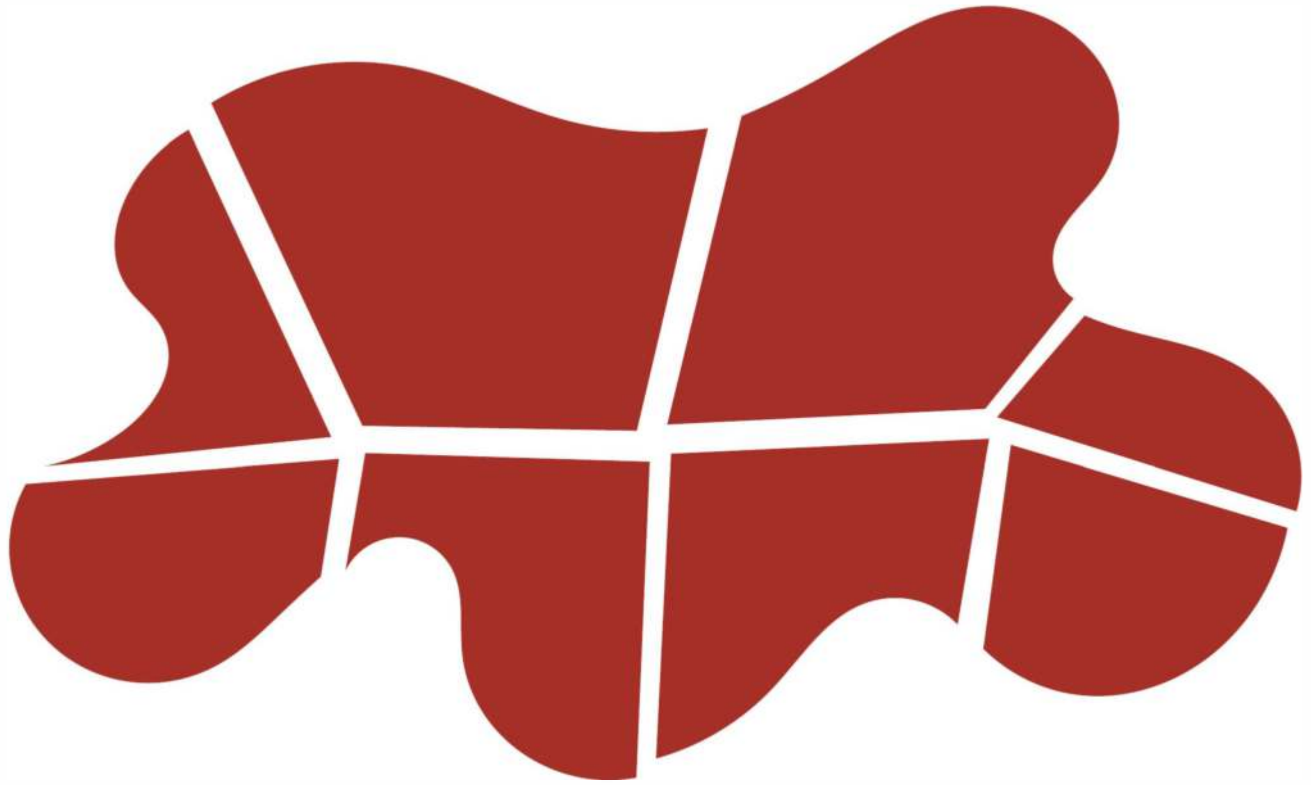
Chunk up the data...

Distributed Data-Parallel: High Level Illustration



Chunk up the data...

Distributed Data-Parallel: High Level Illustration



Distribute it over your cluster of machines.

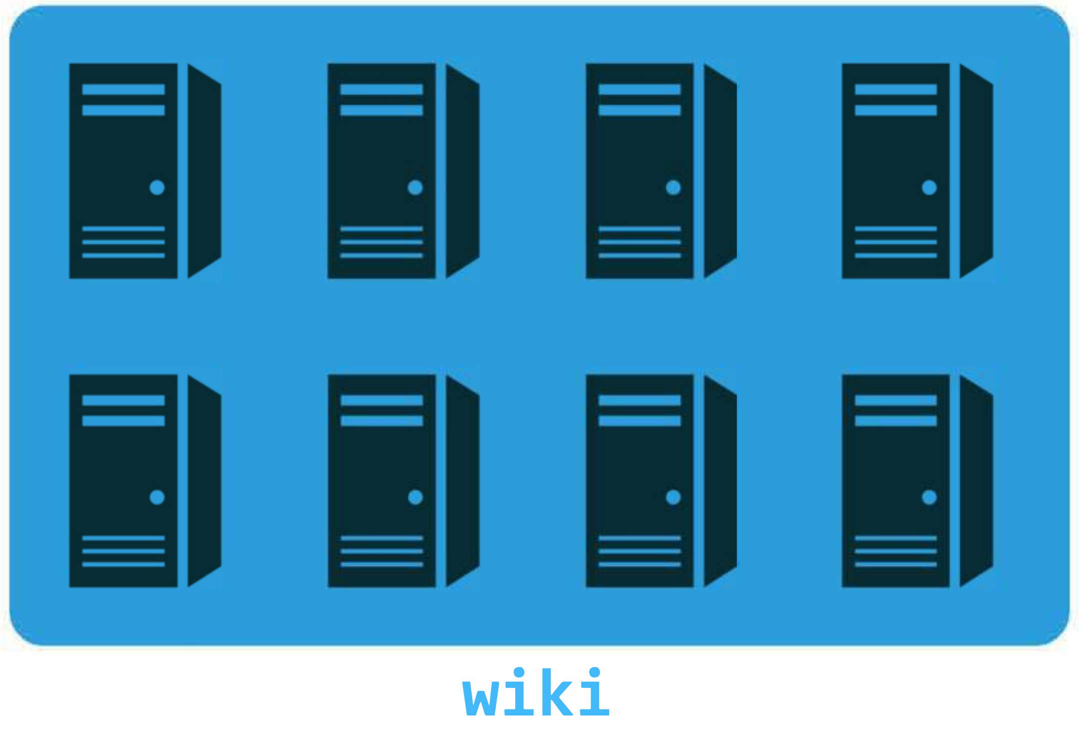
Distributed Data-Parallel: High Level Illustration

Distribute it over your cluster of machines.

Distributed Data-Parallel: High Level Illustration

From there, think of your distributed data like a single collection...

```
val wiki: RDD[WikiArticle] = ...
```



Example:

Transform the text (not titles) of all wiki articles to lowercase.

```
wiki.map {
  article => article.text.toLowerCase
}
```

ÉCOLE POLYTECHNIQUE
FÉDÉRALE DE LAUSANNE

Latency

Big Data Analysis with Scala and Spark

Heather Miller

Data-Parallel Programming

In the Parallel Programming course, we learned:

- Data parallelism on a single multicore/multi-processor machine.
- Parallel collections as an implementation of this paradigm.

Data-Parallel Programming

In the Parallel Programming course, we learned:

- Data parallelism on a single multicore/multi-processor machine.
- Parallel collections as an implementation of this paradigm.

Today:

- Data parallelism in a *distributed setting*.
- Distributed collections abstraction from Apache Spark as an implementation of this paradigm.

Distribution

Distribution introduces important concerns beyond what we had to worry about when dealing with parallelism in the shared memory case:

- *Partial failure:* crash failures of a subset of the machines involved in a distributed computation.
- *Latency:* certain operations have a much higher latency than other operations due to network communication.

Distribution

Distribution introduces important concerns beyond what we had to worry about when dealing with parallelism in the shared memory case:

- *Partial failure:* crash failures of a subset of the machines involved in a distributed computation.
- *Latency:* certain operations have a much higher latency than other operations due to network communication.

⚠ **Latency cannot be masked completely; it will be an important aspect that also impacts the *programming model*.**

Important Latency Numbers

L1 cache reference	0.5ns	
Branch mispredict	5ns	
L2 cache reference	7ns	
Mutex lock/unlock	25ns	
Main memory reference	100ns	
Compress 1K bytes with Zippy	3,000ns	= 3µs
Send 2K bytes over 1Gbps network	20,000ns	= 20µs
SSD random read	150,000ns	= 150µs
Read 1 MB sequentially from	250,000ns	= 250µs
Roundtrip within same datacenter	500,000ns	= 0.5ms
Read 1MB sequentially from SSD	1,000,000ns	= 1ms
Disk seek	10,000,000ns	= 10ms
Read 1MB sequentially from disk	20,000,000ns	= 20ms
Send packet US → Europe → US	150,000,000ns	= 150ms

Original compilation by Jeff Dean & Peter Norvig, w/ contributions by Joe Hellerstein & Erik Meijer

Important Latency Numbers

L1 cache reference	0.5ns	
Branch mispredict	5ns	
L2 cache reference	7ns	
Mutex lock/unlock	25ns	
Main memory reference	100ns	
Compress 1K bytes with Zippy	3,000ns	= 3µs
Send 2K bytes over 1Gbps network	20,000ns	= 20µs
SSD random read	150,000ns	= 150µs
Read 1 MB sequentially from memory	250,000ns	= 250µs
Roundtrip within same datacenter	500,000ns	= 0.5ms
Read 1MB sequentially from SSD	1,000,000ns	= 1ms
Disk seek	10,000,000ns	= 10ms
Read 1MB sequentially from disk	20,000,000ns	= 20ms
Send packet US → Europe → US	150,000,000ns	= 150ms

100x

Original compilation by Jeff Dean & Peter Norvig, w/ contributions by Joe Hellerstein & Erik Meijer

Important Latency Numbers

L1 cache reference	0.5ns	
Branch mispredict	5ns	
L2 cache reference	7ns	
Mutex lock/unlock	25ns	
Main memory reference	100ns	
Compress 1K bytes with Zippy	3,000ns	= 3µs
Send 2K bytes over 1Gbps network	20,000ns	= 20µs
SSD random read	150,000ns	= 150µs
Read 1 MB sequentially from	250,000ns	= 250µs
Roundtrip within same datacenter	500,000ns	= 0.5ms
Read 1MB sequentially from SSD	1,000,000ns	= 1ms
Disk seek	10,000,000ns	= 10ms
Read 1MB sequentially from disk	20,000,000ns	= 20ms
Send packet US → Europe → US	150,000,000ns	= 150ms

1,000,000x SLOWER

Original compilation by Jeff Dean & Peter Norvig, w/ contributions by Joe Hellerstein & Erik Meijer

Important Latency Numbers

L1 cache reference	0.5ns	
Branch mispredict	5ns	
L2 cache reference	7ns	
Mutex lock/unlock	25ns	
Main memory reference	100ns	
Compress 1K bytes with Zippy	3,000ns	= 3µs
Send 2K bytes over 1Gbps network	20,000ns	= 20µs
SSD random read	150,000ns	= 150µs
Read 1 MB sequentially from memory	250,000ns	= 250µs
Roundtrip within same datacenter	500,000ns	= 0.5ms
Read 1MB sequentially from SSD	1,000,000ns	= 1ms
Disk seek	10,000,000ns	= 10ms
Read 1MB sequentially from disk	20,000,000ns	= 20ms
Send packet US → Europe → US	150,000,000ns	= 150ms

memory

disk

network

memory: fastest
disk: slow
network: slowest

Original compilation by Jeff Dean & Peter Norvig, w/ contributions by Joe Hellerstein & Erik Meijer

Latency Numbers Intuitively

To get a better intuition about the *orders-of-magnitude differences* of these numbers, let's **humanize** these durations.

Method: multiply all these durations by a billion.

Then, we can map each latency number to a *human activity*.

Humanized Latency Numbers

Humanized durations grouped by magnitude:

Minute:

```
L1 cache reference           0.5 s         One heart beat (0.5 s)
Branch mispredict            5 s           Yawn
L2 cache reference           7 s           Long yawn
Mutex lock/unlock            25 s          Making a coffee
```

Hour:

```
Main memory reference        100 s         Brushing your teeth
Compress 1K bytes with Zippy 50 min        One episode of a TV show
```

Humanized Latency Numbers

Day:

```
Send 2K bytes over 1 Gbps network   5.5 hr        From lunch to end of work day
```

Week:

```
SSD random read                     1.7 days      A normal weekend
Read 1 MB sequentially from memory  2.9 days      A long weekend
Round trip within same datacenter   5.8 days      A medium vacation
Read 1 MB sequentially from SSD    11.6 days      Waiting for almost 2
                                                  weeks for a delivery
```

More Humanized Latency Numbers

Year:

Disk seek	16.5 weeks	A semester in university
Read 1 MB sequentially from disk	7.8 months	Almost producing a **new** human being
The above 2 together	1 year	

Decade:

Send packet CA->Netherlands->CA	4.8 years	Average time it takes to complete a bachelor's degree

Latency and System Design

Memory

L1 cache reference	**0.5 s**
Main memory reference	**100 s**
Read 1MB sequentially from memory	**2.9 days**

seconds/days

Disk

Disk seek	**16.5 weeks**
Read 1MB sequentially from disk	**7.8 months**

weeks/months

Network

Round trip within same datacenter	**5.8 days**
Send packet US→Eur→US	**4.8 years**

weeks/years

Big Data Processing and Latency?

With some intuition now about how expensive network communication and disk operations can be, one may ask:

How do these latency numbers relate to big data processing?

To answer this question, let's first start with Spark's predecessor, Hadoop.

Hadoop/MapReduce

Hadoop is a widely-used large-scale batch data processing framework. It's an open source implementation of Google's MapReduce.

Hadoop/MapReduce

Hadoop is a widely-used large-scale batch data processing framework. It's an open source implementation of Google's MapReduce.

MapReduce was ground-breaking because it provided:

- a simple API (simple `map` and `reduce` steps)
- ∗∗ **fault tolerance** ∗∗

Hadoop/MapReduce

Hadoop is a widely-used large-scale batch data processing framework. It's an open source implementation of Google's MapReduce.

MapReduce was ground-breaking because it provided:

- a simple API (simple `map` and `reduce` steps)
- ** **fault tolerance** **

Fault tolerance is what made it possible for Hadoop/MapReduce to scale to 100s or 1000s of nodes at all.

Hadoop/MapReduce + Fault Tolerance

Why is this important?

For 100s or 1000s of old commodity machines, likelihood of at least one node failing is **very high** midway through a job.

Hadoop/MapReduce + Fault Tolerance

Why is this important?

For 100s or 1000s of old commodity machines, likelihood of at least one node failing is **very high** midway through a job.

Thus, Hadoop/MapReduce's ability to recover from node failure enabled:

- computations on unthinkably large data sets to succeed to completion.

Hadoop/MapReduce + Fault Tolerance

Why is this important?

For 100s or 1000s of old commodity machines, likelihood of at least one node failing is **very high** midway through a job.

Thus, Hadoop/MapReduce's ability to recover from node failure enabled:

- computations on unthinkably large data sets to succeed to completion.

Fault tolerance + simple API =

At Google, MapReduce made it possible for an average Google software engineer to craft a complex pipeline of map/reduce stages on extremely large data sets.

Why Spark?

Why Spark?

Fault-tolerance in Hadoop/MapReduce comes at a cost.

Between each map and reduce step, in order to recover from potential failures, Hadoop/MapReduce shuffles its data and write intermediate data to disk.

Why Spark?

Fault-tolerance in Hadoop/MapReduce comes at a cost.

Between each map and reduce step, in order to recover from potential failures, Hadoop/MapReduce shuffles its data and write intermediate data to disk.

Remember:

Reading/writing to disk: **100~~0~~x slower** than in-memory
Network communication: **1,000,000x slower** than in-memory

Why Spark?

Spark...

- Retains fault-tolerance
- Different strategy for handling latency (latency significantly reduced!)

Why Spark?

Spark...

- Retains fault-tolerance
- Different strategy for handling latency (latency significantly reduced!)

Achieves this using ideas from functional programming!

Why Spark?

Spark...

- Retains fault-tolerance
- Different strategy for handling latency (latency significantly reduced!)

Achieves this using ideas from functional programming!

Idea: Keep all data **immutable and in-memory**. All operations on data are just functional transformations, like regular Scala collections. Fault tolerance is achieved by replaying functional transformations over original dataset.

Why Spark?

Spark...

- Retains fault-tolerance
- Different strategy for handling latency (latency significantly reduced!)

Achieves this using ideas from functional programming!

Idea: Keep all data **immutable and in-memory**. All operations on data are just functional transformations, like regular Scala collections. Fault tolerance is achieved by replaying functional transformations over original dataset.

Result: Spark has been shown to be 100x more performant than Hadoop, while adding even more expressive APIs.

Latency and System Design (Humanized)

Memory

L1 cache reference	0.5 s
Main memory reference	100 s
Read 1MB sequentially from memory	2.9 days

seconds/days

Disk

Disk seek	16.5 weeks
Read 1MB sequentially from disk	7.8 months

weeks/months

Network

Round trip within same datacenter	5.8 days
Send packet US→Eur→US	4.8 years

weeks/years

Latency and System Design

Memory

L1 cache reference	**0.5 s**
Main memory reference	**100 s**
Read 1MB sequentially from memory	**2.9 days**

seconds/days

Disk

Disk seek	**16.5 weeks**
Read 1MB sequentially from disk	**7.8 months**

weeks/months

Network

Round trip within same datacenter	**5.8 days**
Send packet US→Eur→US	**4.8 years**

weeks/years

Hadoop

Latency and System Design

Memory

L1 cache reference	**0.5 s**
Main memory reference	**100 s**
Read 1MB sequentially from memory	**2.9 days**

seconds/days

Disk

Disk seek	**16.5 weeks**
Read 1MB sequentially from disk	**7.8 months**

weeks/months

Network

Round trip within same datacenter	**5.8 days**
Send packet US→Eur→US	**4.8 years**

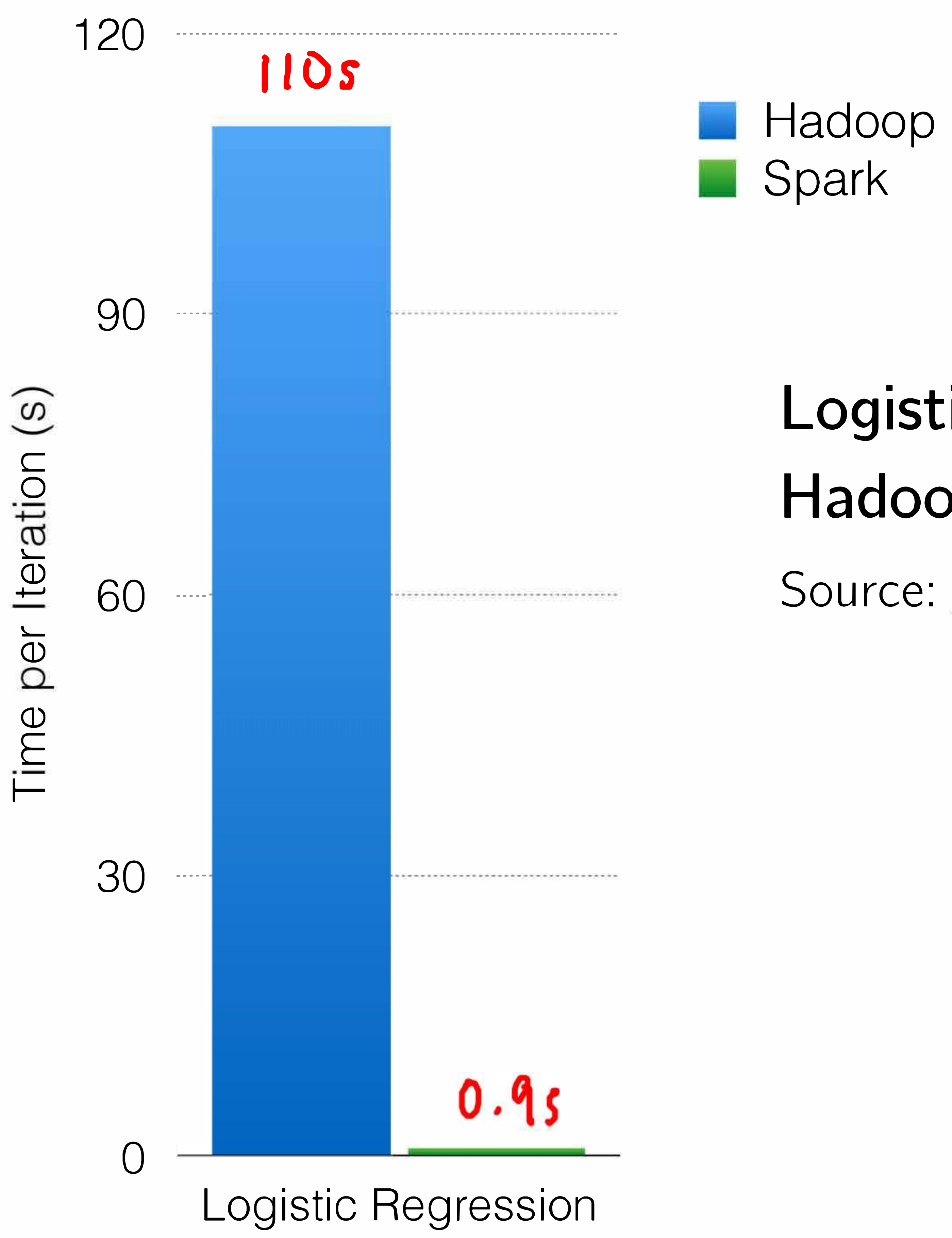
weeks/years

Spark

shift to inmemory

aggressively minimize

Spark versus Hadoop Performance?



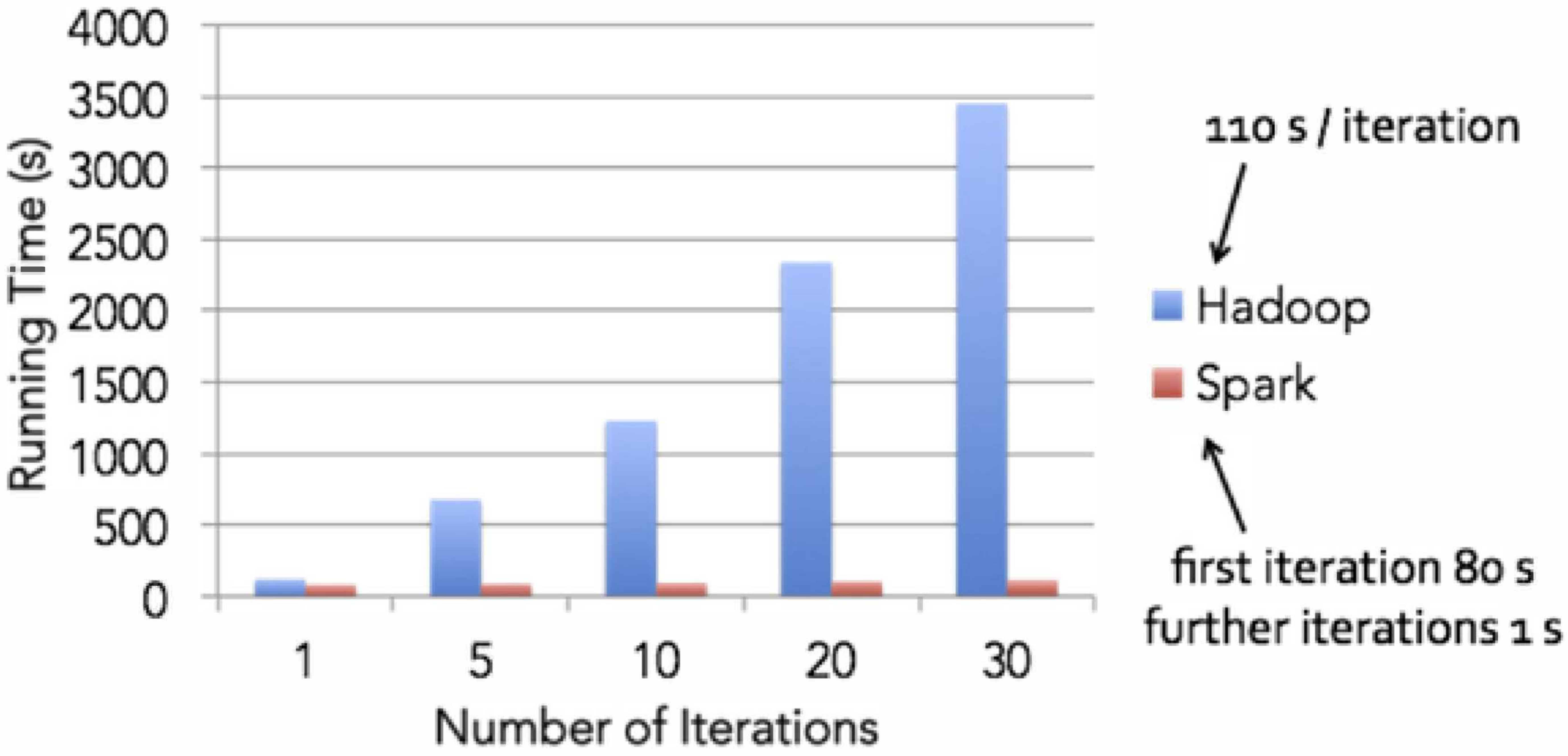
Logistic Regression in Hadoop and Spark

Source: spark.apache.org

Spark versus Hadoop Performance?

Logistic Regression in Hadoop and Spark, more iterations!

Source: https://databricks.com/blog/2014/03/20/apache-spark-a-delight-for-developers.html



110 s / iteration

first iteration 80 s
further iterations 1 s

Hadoop vs Spark Performance, More Intuitively

Day-to-day, these perforamnce improvements can mean the difference between:

Hadoop/MapReduce

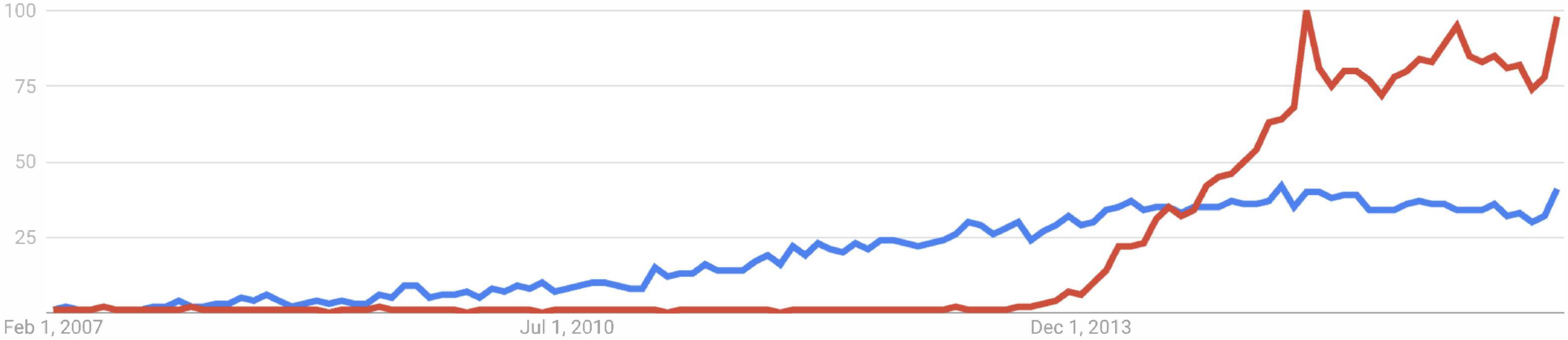
1. start job
2. eat lunch
3. get coffee
4. pick up Kids
5. job completes

Spark

1. start job
2. get coffee
3. job completes

Spark versus Hadoop Popularity?

According to Google Trends, Spark has surpassed Hadoop in popularity.



🟥 Apache Spark

🟦 Apache Hadoop

Google Trends:

Apache Hadoop vs Apache Spark

February 2007 - February 2017

ÉCOLE POLYTECHNIQUE
FÉDÉRALE DE LAUSANNE

Resilient Distributed Datasets(RDDs), Spark's Distributed Collections

Big Data Analysis with Scala and Spark

Heather Miller

Resilient Distributed Datasets (RDDs)

RDDs seem a lot like ***immutable*** sequential or parallel Scala collections.

Resilient Distributed Datasets (RDDs)

RDDs seem a lot like ***immutable*** sequential or parallel Scala collections.

```
abstract class RDD[T] {
  def map[U](f: T => U): RDD[U] = ...
  def flatMap[U](f: T => TraversableOnce[U]): RDD[U] = ...
  def filter(f: T => Boolean): RDD[T] = ...
  def reduce(f: (T, T) => T): T = ...
}
```

Resilient Distributed Datasets (RDDs)

RDDs seem a lot like ***immutable*** sequential or parallel Scala collections.

```
abstract class RDD[T] {
  def map[U](f: T => U): RDD[U] = ...
  def flatMap[U](f: T => TraversableOnce[U]): RDD[U] = ...
  def filter(f: T => Boolean): RDD[T] = ...
  def reduce(f: (T, T) => T): T = ...
}
```

Most operations on RDDs, like Scala's immutable `List`, and Scala's parallel collections, are higher-order functions.

That is, methods that work on RDDs, taking a function as an argument, and which typically return RDDs.

Resilient Distributed Datasets (RDDs)

RDDs seem a lot like ***immutable*** sequential or parallel Scala collections.

Resilient Distributed Datasets (RDDs)

RDDs seem a lot like ***immutable*** sequential or parallel Scala collections.

Combinators on Scala parallel/sequential collections:	**Combinators on RDDs:**
`map`	`map`
`flatMap`	`flatMap`
`filter`	`filter`
`reduce`	`reduce`
`fold`	`fold`
`aggregate`	`aggregate`

Resilient Distributed Datasets (RDDs)

While their signatures differ a bit, their semantics (macroscopically) are the same:

```
map[B](f: A => B): List[B] // Scala List
map[B](f: A => B): RDD[B] // Spark RDD

flatMap[B](f: A => TraversableOnce[B]): List[B] // Scala List
flatMap[B](f: A => TraversableOnce[B]): RDD[B] // Spark RDD

filter(pred: A => Boolean): List[A] // Scala List
filter(pred: A => Boolean): RDD[A] // Spark RDD
```

Resilient Distributed Datasets (RDDs)

While their signatures differ a bit, their semantics (macroscopically) are the same:

```
reduce(op: (A, A) => A): A // Scala List
reduce(op: (A, A) => A): A // Spark RDD

fold(z: A)(op: (A, A) => A): A // Scala List
fold(z: A)(op: (A, A) => A): A // Spark RDD

aggregate[B](z: => B)(seqop: (B, A) => B, combop: (B, B) => B): B // Scala
aggregate[B](z: B)(seqop: (B, A) => B, combop: (B, B) => B): B // Spark RDD
```

Resilient Distributed Datasets (RDDs)

Using RDDs in Spark feels a lot like normal Scala sequential/parallel collections, with the added knowledge that your data is distributed across several machines.

Example:

Given, `val encyclopedia: RDD[String]`, say we want to search all of `encyclopedia` for mentions of EPFL, and count the number of pages that mention EPFL.

Resilient Distributed Datasets (RDDs)

Using RDDs in Spark feels a lot like normal Scala sequential/parallel collections, with the added knowledge that your data is distributed across several machines.

Example:

Given, `val encyclopedia: RDD[String]`, say we want to search all of `encyclopedia` for mentions of EPFL, and count the number of pages that mention EPFL.

```
val result = encyclopedia.filter(page => page.contains(”EPFL”))
                         .count()
```

Example: Word Count

The “Hello, World!” of programming with large-scale data.

RDD[String]

```
// Create an RDD
val rdd = spark.textFile(”hdfs://...”)

val count = ???
```

Example: Word Count

The “Hello, World!” of programming with large-scale data.

```
// Create an RDD
val rdd = spark.textFile(”hdfs://...”)
```

RDD[String] ← words

```
val count = rdd.flatMap(line => line.split(” ”)) // separate lines into words
```

Example: Word Count

The “Hello, World!” of programming with large-scale data.

```
// Create an RDD
val rdd = spark.textFile(”hdfs://...”)

val count = rdd.flatMap(line => line.split(” ”)) // separate lines into words
               .map(word => (word, 1))           // include something to count
```

Example: Word Count

The “Hello, World!” of programming with large-scale data.

```
// Create an RDD
val rdd = spark.textFile(”hdfs://...”)

val count = rdd.flatMap(line => line.split(” ”)) // separate lines into words
               .map(word => (word, 1))           // include something to count
               .reduceByKey(_ + _)               // sum up the 1s in the pairs
```

That’s it.

How to Create an RDD?

RDDs can be created in two ways:

How to Create an RDD?

RDDs can be created in two ways:

- Transforming an existing RDD.
- From a `SparkContext` (or `SparkSession`) object.

How to Create an RDD?

RDDs can be created in two ways:

- **Transforming an existing RDD.**
- From a `SparkContext` (or `SparkSession`) object.

Transforming an existing RDD.
Just like a call to `map` on a `List` returns a new `List`, many higher-order functions defined on `RDD` return a new `RDD`.

How to Create an RDD?

RDDs can be created in two ways:

- Transforming an existing RDD.
- **From a `SparkContext` (or `SparkSession`) object.**

!!! !!!

From a `SparkContext` (or `SparkSession`) object.
The `SparkContext` object (renamed `SparkSession`) can be thought of as your handle to the Spark cluster. It represents the connection between the Spark cluster and your running application. It defines a handful of methods which can be used to create and populate a new RDD:

- `parallelize`: convert a local Scala collection to an RDD.
- `textFile`: read a text file from HDFS or a local file system and return an RDD of `String`

ÉCOLE POLYTECHNIQUE
FÉDÉRALE DE LAUSANNE

Transformations and Actions

operations o
RDDs.

Big Data Analysis with Scala and Spark

Heather Miller

Transformations and Actions

Recall ***transformers*** and ***accessors*** from Scala sequential and parallel collections.

Transformations and Actions

Recall ***transformers*** and ***accessors*** from Scala sequential and parallel collections.

Transformers. Return new collections as results. (Not single values.)
Examples: `map`, `filter`, `flatMap`, `groupBy`

```
map(f: A => B): Traversable[B]
```

Transformations and Actions

Recall ***transformers*** and ***accessors*** from Scala sequential and parallel collections.

Transformers. Return new collections as results. (Not single values.)
Examples: `map`, `filter`, `flatMap`, `groupBy`

```
map(f: A => B): Traversable[B]
```

Accessors: Return single values as results. (Not collections.)
Examples: `reduce`, `fold`, `aggregate`.

```
reduce(op: (A, A) => A): A
```

Transformations and Actions

Similarly, Spark defines ***transformations*** and ***actions*** on RDDs.

They seem similar to ~~transformers~~ and ~~accessors~~, but there are some important differences.

Transformations. Return new ~~collections~~ RDDs as results.

Actions. Compute a result based on an RDD, and either returned or saved to an external storage system (e.g., HDFS).

Transformations and Actions

Similarly, Spark defines ***transformations*** and ***actions*** on RDDs.

They seem similar to ~~transformers~~ and ~~accessors~~, but there are some important differences.

!!! **Transformations.** Return new ~~collections~~ RDDs as results.

They are lazy, their result RDD is not immediately computed.

!!! **Actions.** Compute a result based on an RDD, and either returned or saved to an external storage system (e.g., HDFS).

They are eager, their result is immediately computed.

Transformations and Actions

Similarly, Spark defines ***transformations*** and ***actions*** on RDDs.

They seem similar to ~~transformers~~ and ~~accessors~~, but there are some important differences.

Transformations. Return new ~~collections~~ RDDs as results.

They are lazy, their result RDD is not immediately computed.

Actions. Compute a result based on an RDD, and either returned or saved to an external storage system (e.g., HDFS).

They are eager, their result is immediately computed.

!!!

Laziness/eagerness is how we can limit network communication using the programming model.

!!!

Example

Consider the following simple example:

sc → SparkContext

```
val largeList: List[String] = ...
val wordsRdd = sc.parallelize(largeList)
val lengthsRdd = wordsRdd.map(_.length)
```

RDD[String]

RDD[Int]

What has happened on the cluster at this point?

Example

Consider the following simple example:

```scala
val largeList: List[String] = ...
val wordsRdd = sc.parallelize(largeList)
val lengthsRdd = wordsRdd.map(_.length)
```

What has happened on the cluster at this point?

Nothing. Execution of `map` (a transformation) is deferred.

To kick off the computation and wait for its result...

Example

Consider the following simple example:

```
val largeList: List[String] = ...
val wordsRdd = sc.parallelize(largeList)
val lengthsRdd = wordsRdd.map(_.length)
val totalChars = lengthsRdd.reduce(_ + _)
```

...we can add an action

Common Transformations in the Wild

LAZY!!

map

`map[B](f: A => B): RDD[B]` ←

Apply function to each element in the RDD and retrun an RDD of the result.

flatMap

`flatMap[B](f: A => TraversableOnce[B]): RDD[B]` ←

Apply a function to each element in the RDD and return an RDD of the contents of the iterators returned.

filter

`filter(pred: A => Boolean): RDD[A]` ←

Apply predicate function to each element in the RDD and return an RDD of elements that have passed the predicate condition, `pred`.

distinct

`distinct(): RDD[B]` ←

Return RDD with duplicates removed.

Common Actions in the Wild

EAGER!

collect	**collect(): Array[T]** ← Return all elements from RDD.
count	**count(): Long** ← Return the number of elements in the RDD.
take	**take(num: Int): Array[T]** ← Return the first num elements of the RDD.
reduce	**reduce(op: (A, A) => A): A** ← Combine the elements in the RDD together using op function and return result.
foreach	**foreach(f: T => Unit): Unit** ← Apply function to each element in the RDD.

Another Example

Let's assume that we have an `RDD[String]` which contains gigabytes of logs collected over the previous year. Each element of this RDD represents one line of logging.

Assuming that dates come in the form, `YYYY-MM-DD:HH:MM:SS`, and errors are logged with a prefix that includes the word "error"...

How would you determine the number of errors that were logged in December 2016?

```scala
val lastYearsLogs: RDD[String] = ...
```

Another Example

Let’s assume that we have an `RDD[String]` which contains gigabytes of logs collected over the previous year. Each element of this RDD represents one line of logging.

Assuming that dates come in the form, `YYYY-MM-DD:HH:MM:SS`, and errors are logged with a prefix that includes the word “error”...

How would you determine the number of errors that were logged in December 2016?

```
val lastYearsLogs: RDD[String] = ...
val numDecErrorLogs
  = lastYearsLogs.filter(lg => lg.contains(”2016-12”) && lg.contains(”error”))
                 .count()
```

Benefits of Laziness for Large-Scale Data

Spark computes RDDs the first time they are used in an action.

This helps when processing large amounts of data.

filter
↓
take

Example:

```
val lastYearsLogs: RDD[String] = ...
val firstLogsWithErrors = lastYearsLogs.filter(_.contains(”ERROR”)).take(10)
```

The execution of `filter` is deferred until the `take` action is applied.

Spark leverages this by analyzing and optimizing the **chain of operations** before executing it.

Spark will not compute intermediate RDDs. Instead, as soon as 10 elements of the filtered RDD have been computed, `firstLogsWithErrors` is done. At this point Spark stops working, saving time and space computing elements of the unused result of `filter`.

Transformations on Two RDDs

LAZY!!

rdd1 rdd2

Val rdd3 = rdd1.union(rdd2)

RDDs also support set-like operations, like `union` and `intersection`.

Two-RDD transformations combine two RDDs are combined into one.

`union`

`union(other: RDD[T]): RDD[T]` ←

Return an RDD containing elements from both RDDs.

`intersection`

`intersection(other: RDD[T]): RDD[T]` ←

Return an RDD containing elements only found in both RDDs.

`subtract`

`subtract(other: RDD[T]): RDD[T]` ←

Return an RDD with the contents of the other RDD removed.

`cartesian`

`cartesian[U](other: RDD[U]): RDD[(T, U)]` ←

Cartesian product with the other RDD.

Other Useful RDD Actions

EAGER!!

RDDs also contain other important actions unrelated to regular Scala collections, but which are useful when dealing with distributed data.

`takeSample`

`takeSample(withRepl: Boolean, num: Int): Array[T]`
Return an array with a random sample of num elements of the dataset, with or without replacement.

`takeOrdered`

`takeOrdered(num: Int)(implicit ord: Ordering[T]): Array[T]`
Return the first n elements of the RDD using either their natural order or a custom comparator.

`saveAsTextFile`

`saveAsTextFile(path: String): Unit`
Write the elements of the dataset as a text file in the local filesystem or HDFS.

`saveAsSequenceFile`

`saveAsSequenceFile(path: String): Unit`
Write the elements of the dataset as a Hadoop SequenceFile in the local filesystem or HDFS.

ÉCOLE POLYTECHNIQUE
FÉDÉRALE DE LAUSANNE

Evaluation in Spark: Unlike Scala Collections!

Big Data Analysis with Scala and Spark

Heather Miller

Why is Spark Good for Data Science?

Why is Spark Good for Data Science?

Let’s start by recapping some major themes from previous sessions:

- We learned the difference between transformations and actions.
  - **Transformations:** Deferred/lazy
  - **Actions:** Eager, kick off staged transformations.
- We learned that latency makes a big difference; too much latency wastes the time of the data analyst.
  - **In-memory computation:** Significantly lower latencies (several orders of magnitude!)

Why is Spark Good for Data Science?

Let's start by recapping some major themes from previous sessions:

- We learned the difference between transformations and actions.
  - **Transformations:** Deferred/lazy
  - **Actions:** Eager, kick off staged transformations.
- We learned that latency makes a big difference; too much latency wastes the time of the data analyst.
  - **In-memory computation:** Significantly lower latencies (several orders of magnitude!)

Why do you think Spark is good for data science?

Why is Spark Good for Data Science?

Let's start by recapping some major themes from previous sessions:

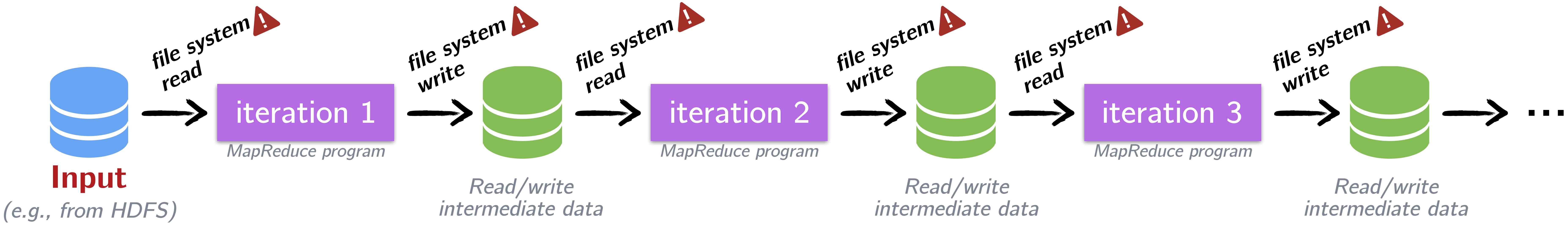
- We learned the difference between transformations and actions.
  - **Transformations:** Deferred/lazy
  - **Actions:** Eager, kick off staged transformations.
- We learned that latency makes a big difference; too much latency wastes the time of the data analyst.
  - **In-memory computation:** Significantly lower latencies (several orders of magnitude!)

Why do you think Spark is good for data science?

Hint: Most data science problems involve iteration.

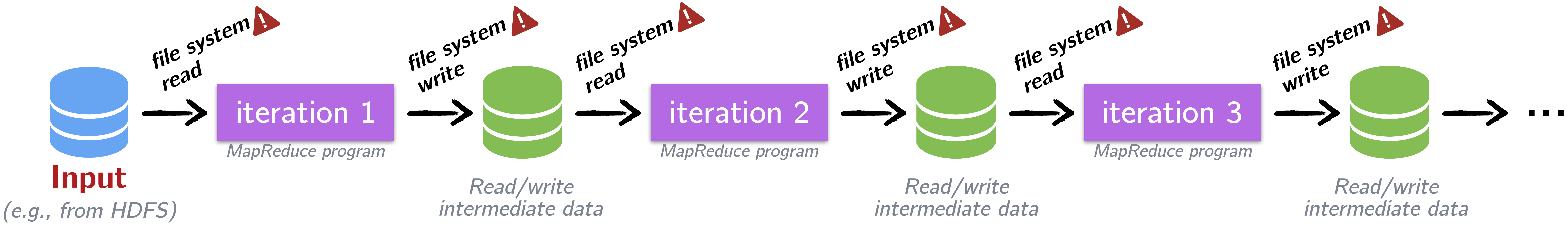
Iteration and Big Data Processing

Iteration in Hadoop:



Iteration and Big Data Processing

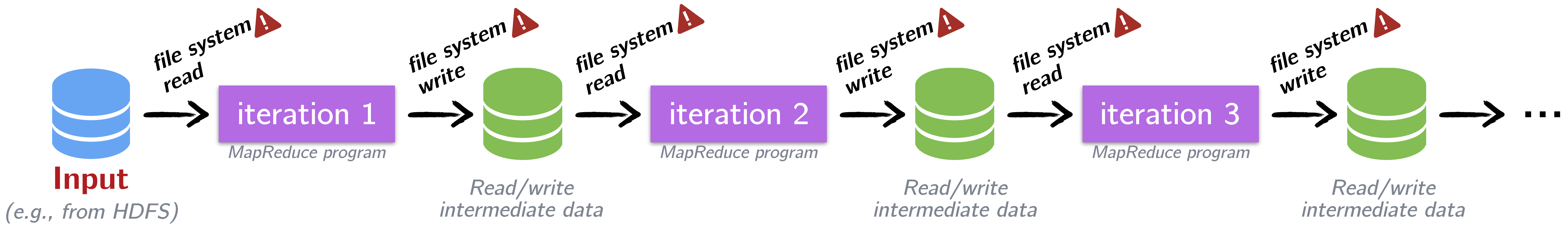
Iteration in Hadoop:



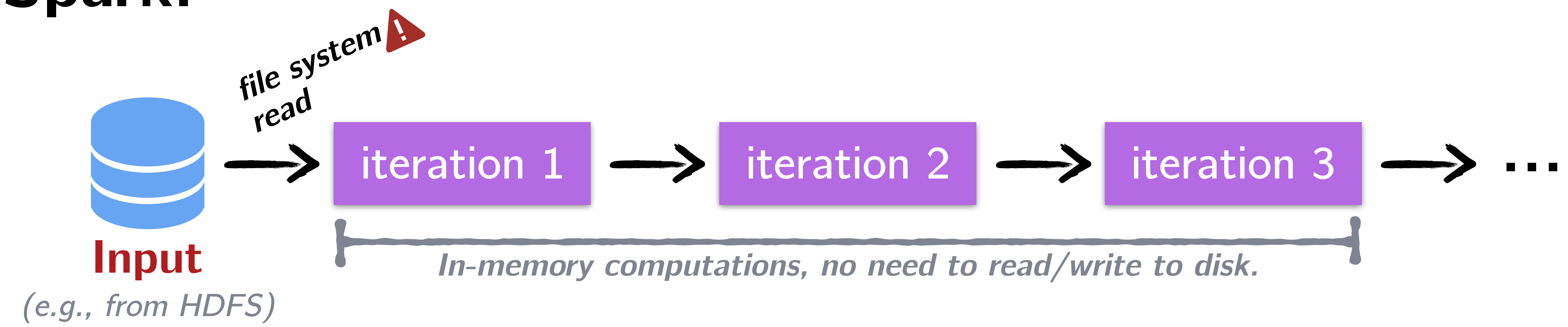
>90% of time in IO that Spark can avoid.

Iteration and Big Data Processing

Iteration in Hadoop:

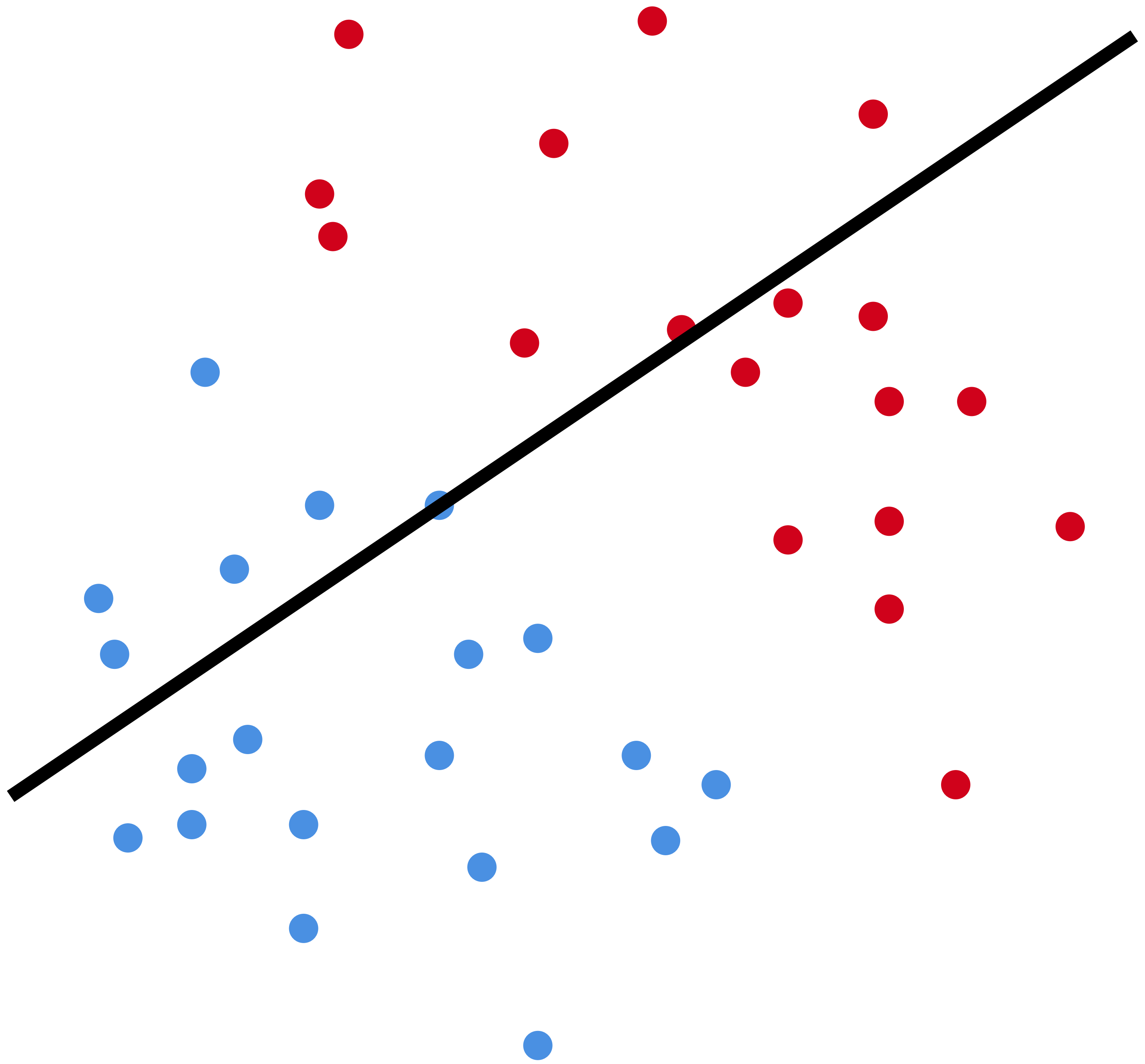


Iteration in Spark:



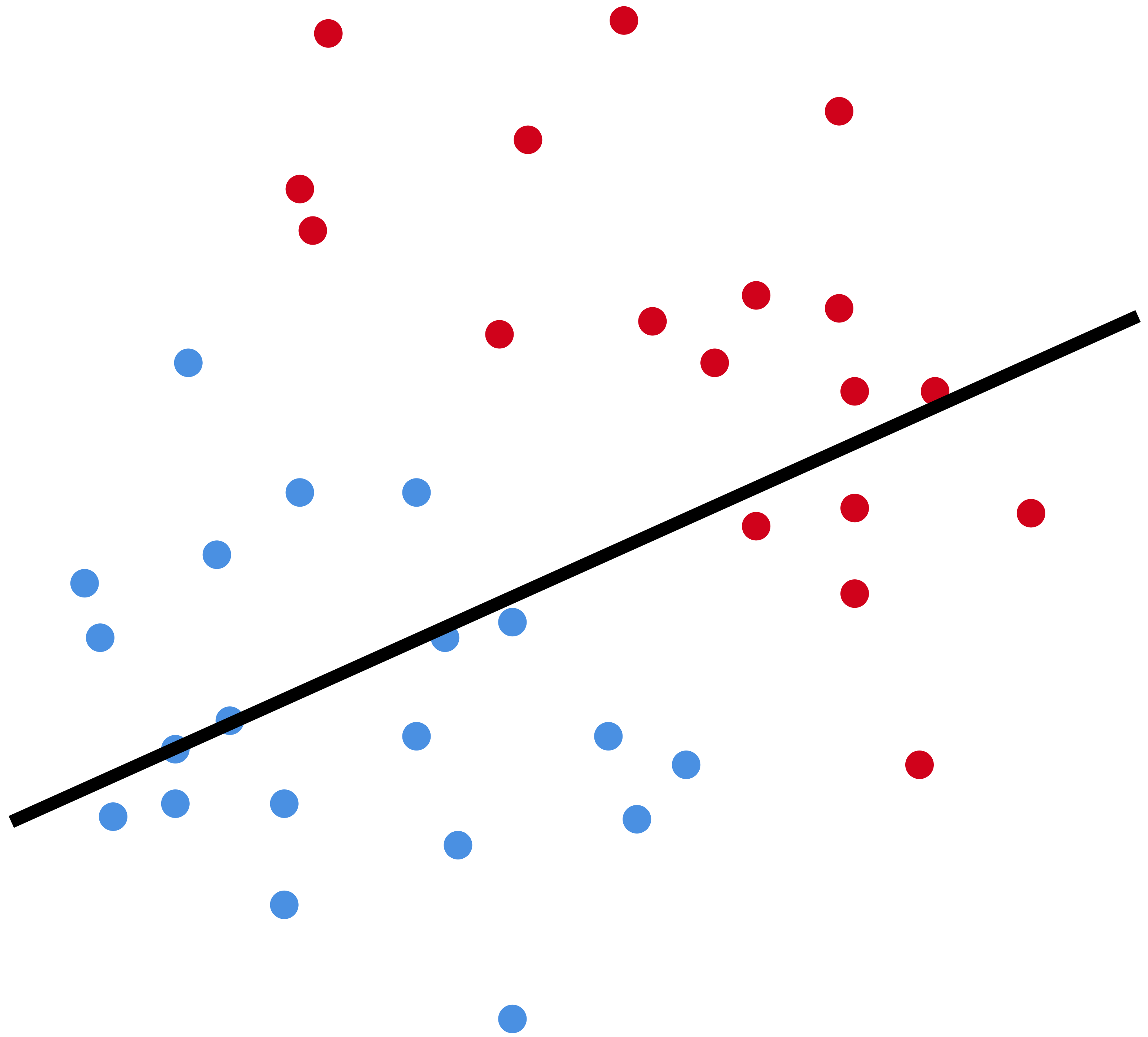
Iteration, Example: Logistic Regression

Logistic regression is an iterative algorithm typically used for classification. Like other classification algorithms, the classifier's weights are iteratively updated based on a training dataset.



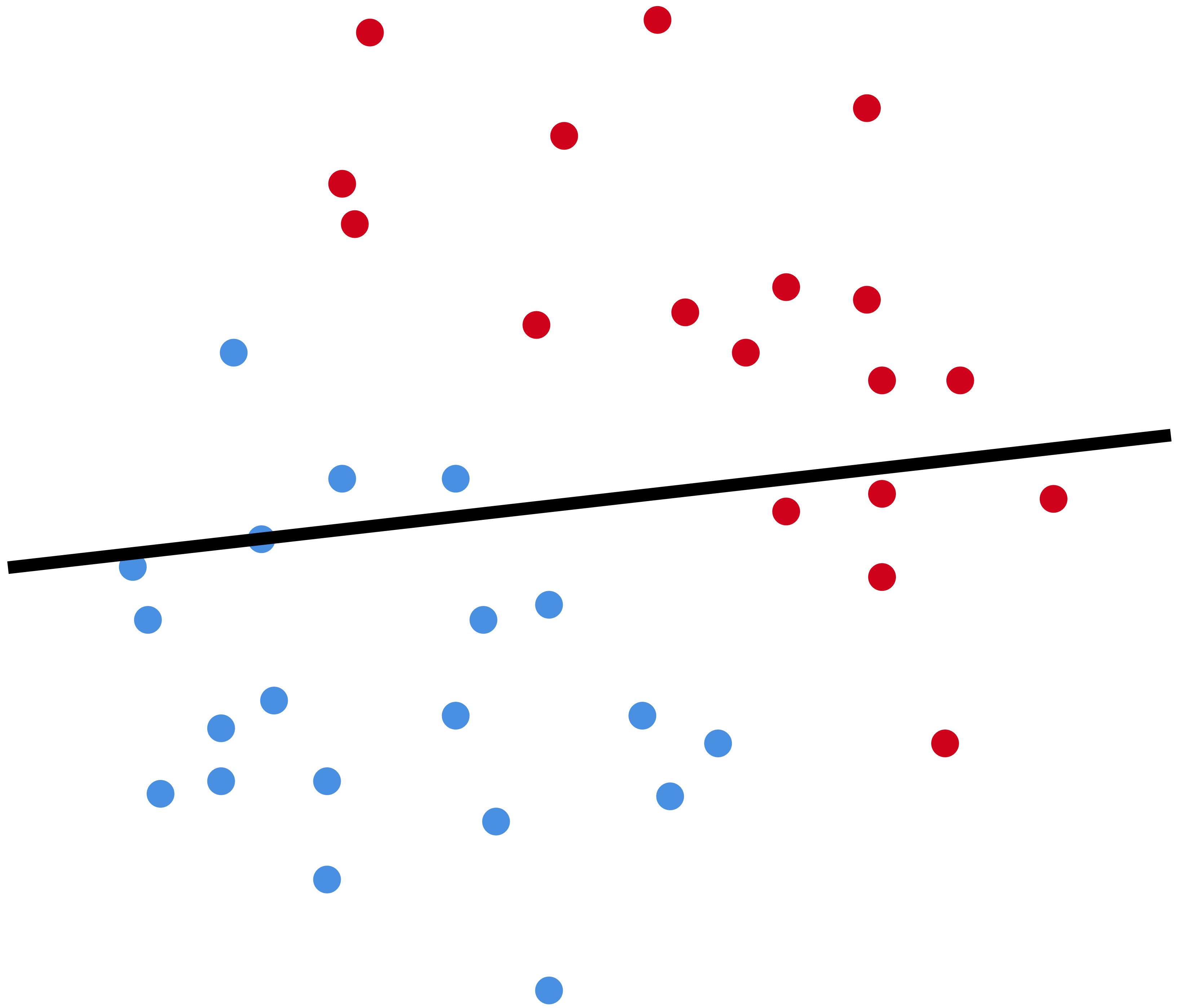
Iteration, Example: Logistic Regression

Logistic regression is an iterative algorithm typically used for classification. Like other classification algorithms, the classifier's weights are iteratively updated based on a training dataset.



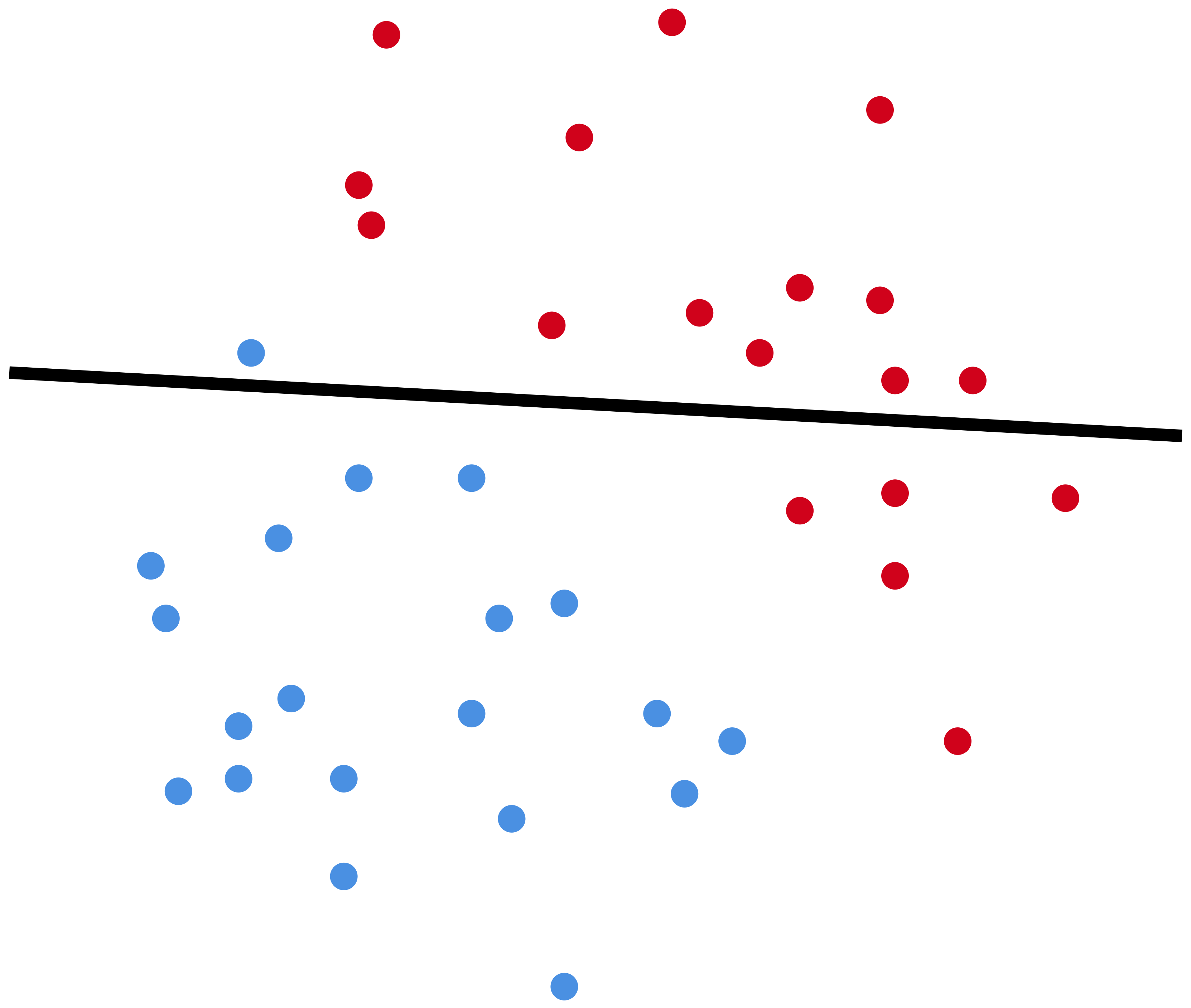
Iteration, Example: Logistic Regression

Logistic regression is an iterative algorithm typically used for classification. Like other classification algorithms, the classifier's weights are iteratively updated based on a training dataset.



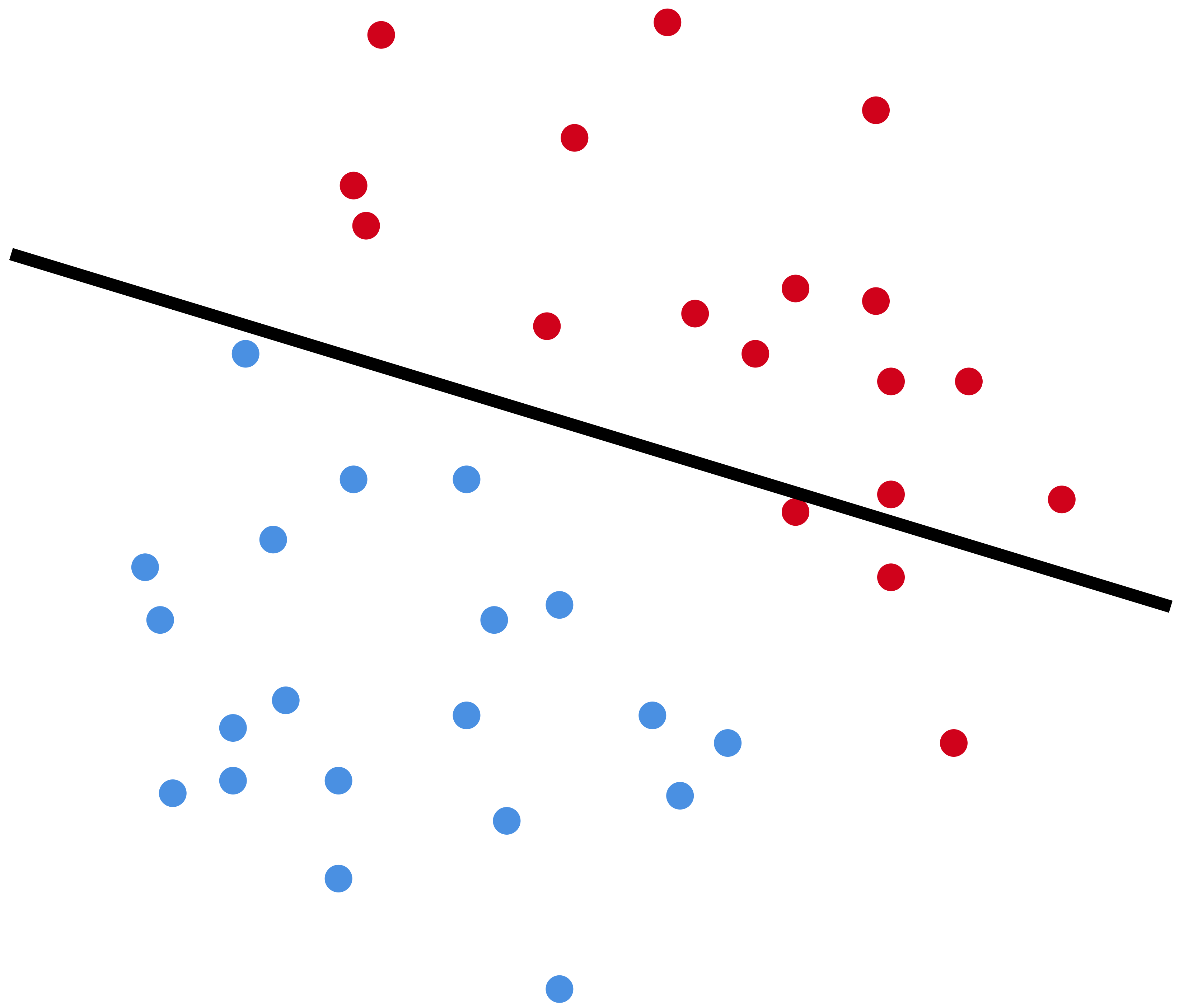
Iteration, Example: Logistic Regression

Logistic regression is an iterative algorithm typically used for classification. Like other classification algorithms, the classifier's weights are iteratively updated based on a training dataset.



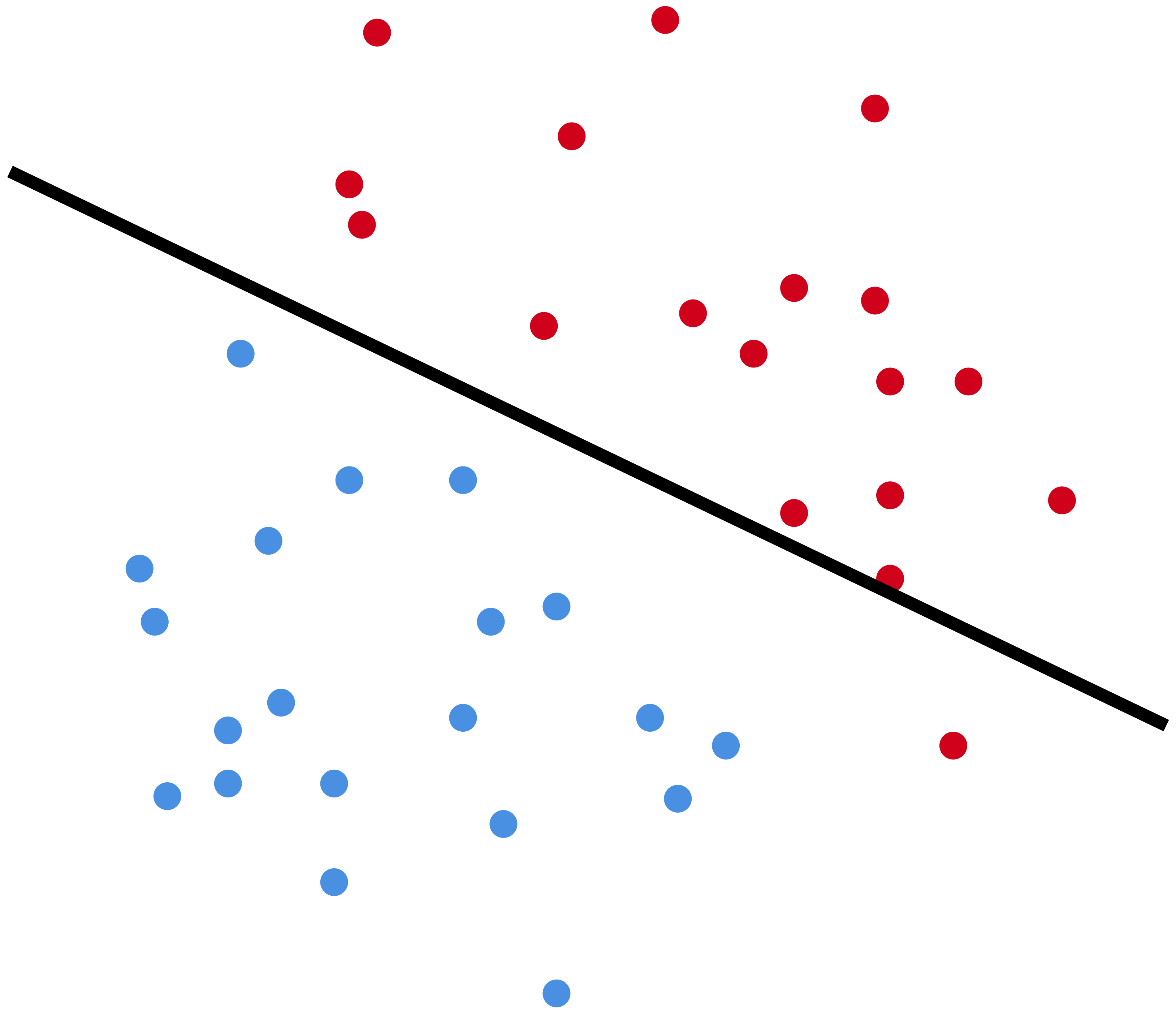
Iteration, Example: Logistic Regression

Logistic regression is an iterative algorithm typically used for classification. Like other classification algorithms, the classifier's weights are iteratively updated based on a training dataset.



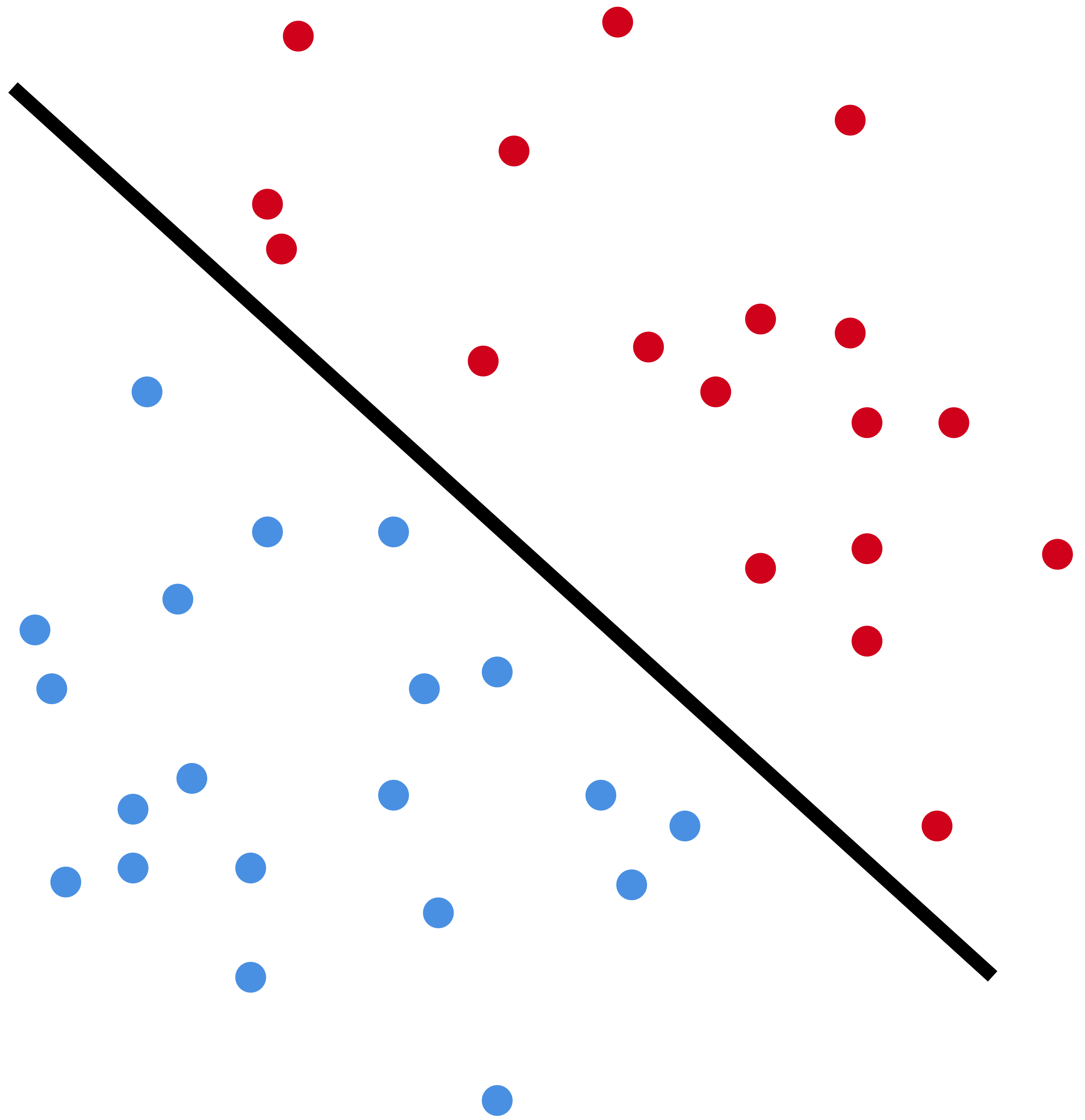
Iteration, Example: Logistic Regression

Logistic regression is an iterative algorithm typically used for classification. Like other classification algorithms, the classifier's weights are iteratively updated based on a training dataset.



Iteration, Example: Logistic Regression

Logistic regression is an iterative algorithm typically used for classification. Like other classification algorithms, the classifier's weights are iteratively updated based on a training dataset.



Iteration, Example: Logistic Regression

Logistic regression is an iterative algorithm typically used for classification. Like other classification algorithms, the classifier's weights are iteratively updated based on a training dataset.

$$w \leftarrow w - \alpha \cdot \sum_{i=1}^{n} g(w; x_i, y_i)$$

Iteration, Example: Logistic Regression

Logistic regression is an iterative algorithm typically used for classification. Like other classification algorithms, the classifier's weights are iteratively updated based on a training dataset.

$$w \leftarrow w - \alpha \cdot \sum_{i=1}^{n} g(w; x_i, y_i)$$

Logistic regression can be implemented in Spark in a straightforward way:

case class Point(x: Double, y: Double)

```
val points = sc.textFile(...).map(parsePoint)
var w = Vector.zeros(d)
for (i <- 1 to numIterations) {
  val gradient = points.map { p =>
    (1 / (1 + exp(-p.y * w.dot(p.x)))- 1) * p.y * p.y
  }.reduce(_ + _)
  w -= alpha * gradient
}
```

What's going on in this code snippet?

Iteration, Example: Logistic Regression

Logistic regression is an iterative algorithm typically used for classification. Like other classification algorithms, the classifier's weights are iteratively updated based on a training dataset.

```
val points = sc.textFile(...).map(parsePoint)
var w = Vector.zeros(d)
for (i <- 1 to numIterations) {
  val gradient = points.map { p =>
    (1 / (1 + exp(-p.y * w.dot(p.x)))- 1) * p.y * p.y
  }.reduce(_ + _)
  w -= alpha * gradient
}
```

`points` is being re-evaluated upon every iteration!
That's unnecessary! What can we do about this?

Caching and Persistence

By default, RDDs are recomputed each time you run an action on them. This can be expensive (in time) if you need to use a dataset more than once.

Spark allows you to control what is cached in memory.

Caching and Persistence

By default, RDDs are recomputed each time you run an action on them. This can be expensive (in time) if you need to use a dataset more than once.

Spark allows you to control what is cached in memory.

To tell Spark to cache an RDD in memory, simply call `persist()` or `cache()` on it.

Caching and Persistence

By default, RDDs are recomputed each time you run an action on them. This can be expensive (in time) if you need to use a dataset more than once.

Spark allows you to control what is cached in memory.

```
val lastYearsLogs: RDD[String] = ...
val logsWithErrors = lastYearsLogs.filter(_.contains(”ERROR”)).persist()
val firstLogsWithErrors = logsWithErrors.take(10)
```

Here, we *cache* `logsWithErrors` in memory.

After `firstLogsWithErrors` is computed, Spark will store the contents of `logsWithErrors` for faster access in future operations if we would like to reuse it.

Caching and Persistence

By default, RDDs are recomputed each time you run an action on them. This can be expensive (in time) if you need to use a dataset more than once.

Spark allows you to control what is cached in memory.

```
val lastYearsLogs: RDD[String] = ...
val logsWithErrors = lastYearsLogs.filter(_.contains(”ERROR”)).persist()
val firstLogsWithErrors = logsWithErrors.take(10)
val numErrors = logsWithErrors.count() // faster
```

Now, computing the `count` on `logsWithErrors` is much faster.

Back to Our Logistic Regression Example

Logistic regression is an iterative algorithm typically used for classification. Like other classification algorithms, the classifier's weights are iteratively updated based on a training dataset.

```
val points = sc.textFile(...).map(parsePoint).persist()
var w = Vector.zeros(d)
for (i <- 1 to numIterations) {
  val gradient = points.map { p =>
    (1 / (1 + exp(-p.y * w.dot(p.x)))- 1) * p.y * p.y
  }.reduce(_ + _)
  w -= alpha * gradient
}
```

Now, `points` is evaluated once and and is cached in memory. It is then re-used on each iteration.

Caching and Persistence

There are many ways to configure how your data is persisted.

Possible to persist data set:

- in memory as regular Java objects
- on disk as regular Java objects
- in memory as serialized Java objects (more compact)
- on disk as serialized Java objects (more compact)
- both in memory and on disk (spill over to disk to avoid re-computation)

`cache`( )
Shorthand for using the default storage level, which is in memory only as regular Java objects.

`persist`
Persistence can be customized with this method. Pass the storage level you'd like as a parameter to `persist`.

Caching and Persistence

Storage levels. Other ways to control how Spark stores objects.

Level	*Space used*	*CPU time*	*In memory*	*On disk*
`MEMORY_ONLY`	High	Low	Y	N
`MEMORY_ONLY_SER`	Low	High	Y	N
`MEMORY_AND_DISK`*	High	Medium	Some	Some
`MEMORY_AND_DISK_SER`†	Low	High	Some	Some
`DISK_ONLY`	Low	High	N	Y

* Spills to disk if there is too much data to fit in memory

† Spills to disk if there is too much data to fit in memory. Stores serialized representation in memory.

Caching and Persistence

Storage levels. Other ways to control how Spark stores objects.

Level	*Space used*	*CPU time*	*In memory*	*On disk*
MEMORY_ONLY	**High**	**Low**	**Y**	**N**
MEMORY_ONLY_SER	Low	High	Y	N
MEMORY_AND_DISK*	High	Medium	Some	Some
MEMORY_AND_DISK_SER†	Low	High	Some	Some
DISK_ONLY	Low	High	N	Y

Default

* Spills to disk if there is too much data to fit in memory

† Spills to disk if there is too much data to fit in memory. Stores serialized representation in memory.

RDDs Look Like Collections, But Behave Totally Differently

Key takeaway:

Despite similar-looking API to Scala Collections,
the deferred semantics of Spark's RDDs are very unlike Scala Collections.

RDDs Look Like Collections, But Behave Totally Differently

Key takeaway:

Despite similar-looking API to Scala Collections,
the deferred semantics of Spark's RDDs are very unlike Scala Collections.

Due to:

- the lazy semantics of RDD transformation operations (`map`, `flatMap`, `filter`),
- and users' implicit reflex to assume collections are eagerly evaluated...

...One of the most common performance bottlenecks of newcomers to Spark arises from unknowingly re-evaluating several transformations when caching could be used.

RDDs Look Like Collections, But Behave Totally Differently

Key takeaway:

Despite similar-looking API to Scala Collections,
the deferred semantics of Spark's RDDs are very unlike Scala Collections.

Due to:

- the lazy semantics of RDD transformation operations (`map`, `flatMap`, `filter`),
- and users' implicit reflex to assume collections are eagerly evaluated...

...One of the most common performance bottlenecks of newcomers to Spark arises from unknowingly re-evaluating several transformations when caching could be used.

Don't make this mistake in your programming assignments.

Restating the Benefits of Laziness for Large-Scale Data

While many users struggle with the lazy semantics of RDDs at first, it's helpful to remember the ways in which these semantics are helpful in the face of large-scale distributed computing.

Example #1:

```
val lastYearsLogs: RDD[String] = ...
val firstLogsWithErrors = lastYearsLogs.filter(_.contains(”ERROR”)).take(10)
```

Restating the Benefits of Laziness for Large-Scale Data

While many users struggle with the lazy semantics of RDDs at first, it's helpful to remember the ways in which these semantics are helpful in the face of large-scale distributed computing.

Example #1:

```
val lastYearsLogs: RDD[String] = ...
val firstLogsWithErrors = lastYearsLogs.filter(_.contains(”ERROR”)).take(10)
```

The execution of `filter` is deferred until the `take` action is applied.

Spark leverages this by analyzing and optimizing the **chain of operations** before executing it.

Spark will not compute intermediate RDDs. Instead, as soon as 10 elements of the filtered RDD have been computed, `firstLogsWithErrors` is done. At this point Spark stops working, saving time and space computing elements of the unused result of `filter`.

Restating the Benefits of Laziness for Large-Scale Data

While many users struggle with the lazy semantics of RDDs at first, it’s helpful to remember the ways in which these semantics are helpful in the face of large-scale distributed computing.

Example #2:

```
val lastYearsLogs: RDD[String] = ...
val numErrors = lastYearsLogs.map(_.lowercase)
                             .filter(_.contains(”error”))
                             .count()
```

Restating the Benefits of Laziness for Large-Scale Data

While many users struggle with the lazy semantics of RDDs at first, it’s helpful to remember the ways in which these semantics are helpful in the face of large-scale distributed computing.

Example #2:

```scala
val lastYearsLogs: RDD[String] = ...
val numErrors = lastYearsLogs.map(_.lowercase)
                             .filter(_.contains(”error”))
                             .count()
```

Lazy evaluation of these transformations allows Spark to *stage* computations. That is, Spark can make important optimizations to the the **chain of operations** before execution.

For example, after calling `map` and `filter`, Spark knows that it can avoid doing multiple passes through the data. That is, Spark can traverse through the RDD once, computing the result of `map` and `filter` in this single pass, before returning the resulting count.

ÉCOLE POLYTECHNIQUE
FÉDÉRALE DE LAUSANNE

Cluster Toplogy Matters!

Big Data Analysis with Scala and Spark

Heather Miller

Example 1: A Simple println

Let's start with an example. Assume we have an RDD populated with Person objects:

```
case class Person(name: String, age: Int)
```

What does the following code snippet do?

```
val people: RDD[Person] = ...
people.foreach(println)
```

What happens?

Example 2: A Simple `take`

What about here? Assume we have an RDD populated with the same definition of `Person` objects:

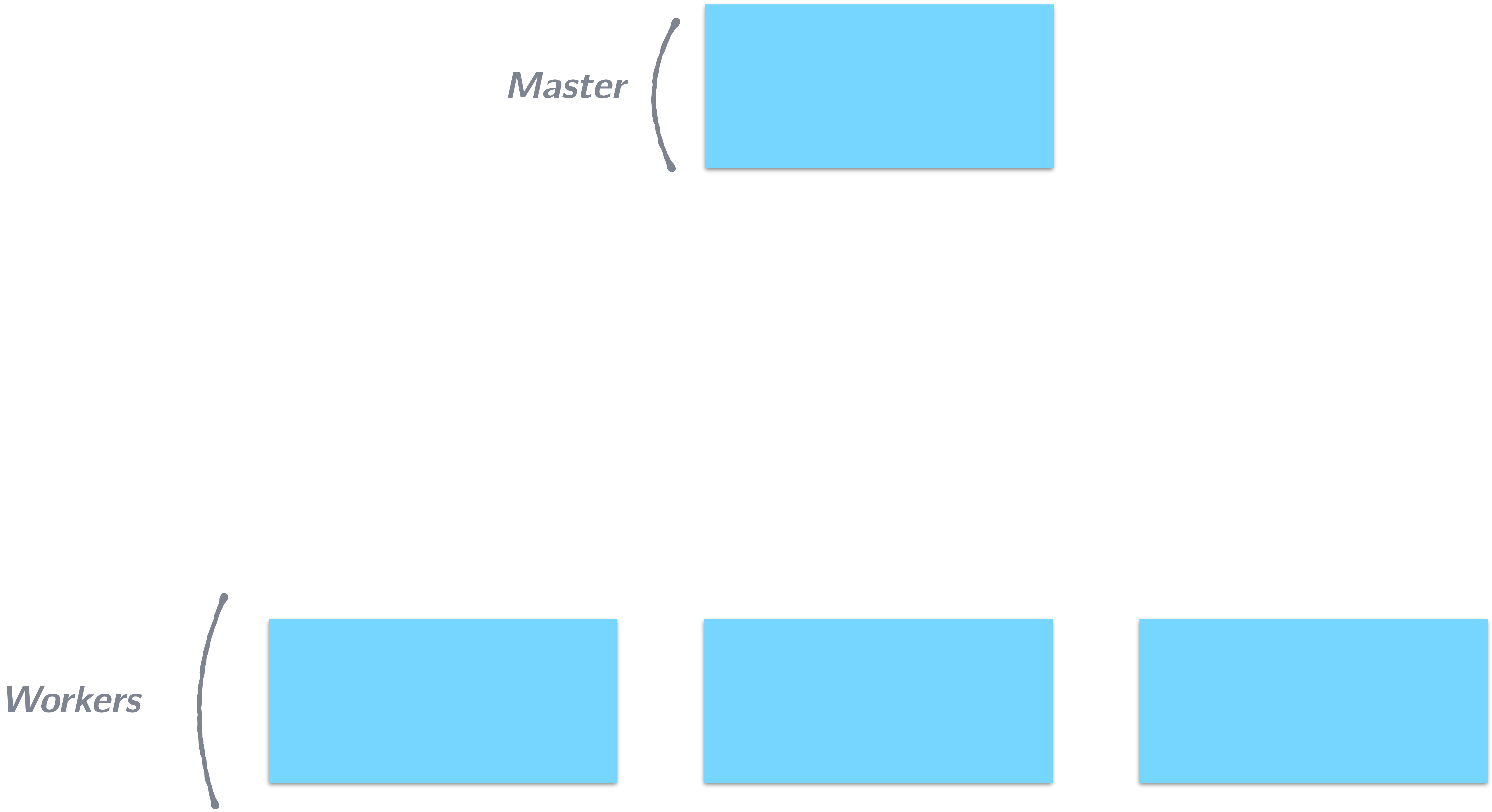
```
case class Person(name: String, age: Int)
```

What does the following code snippet do?

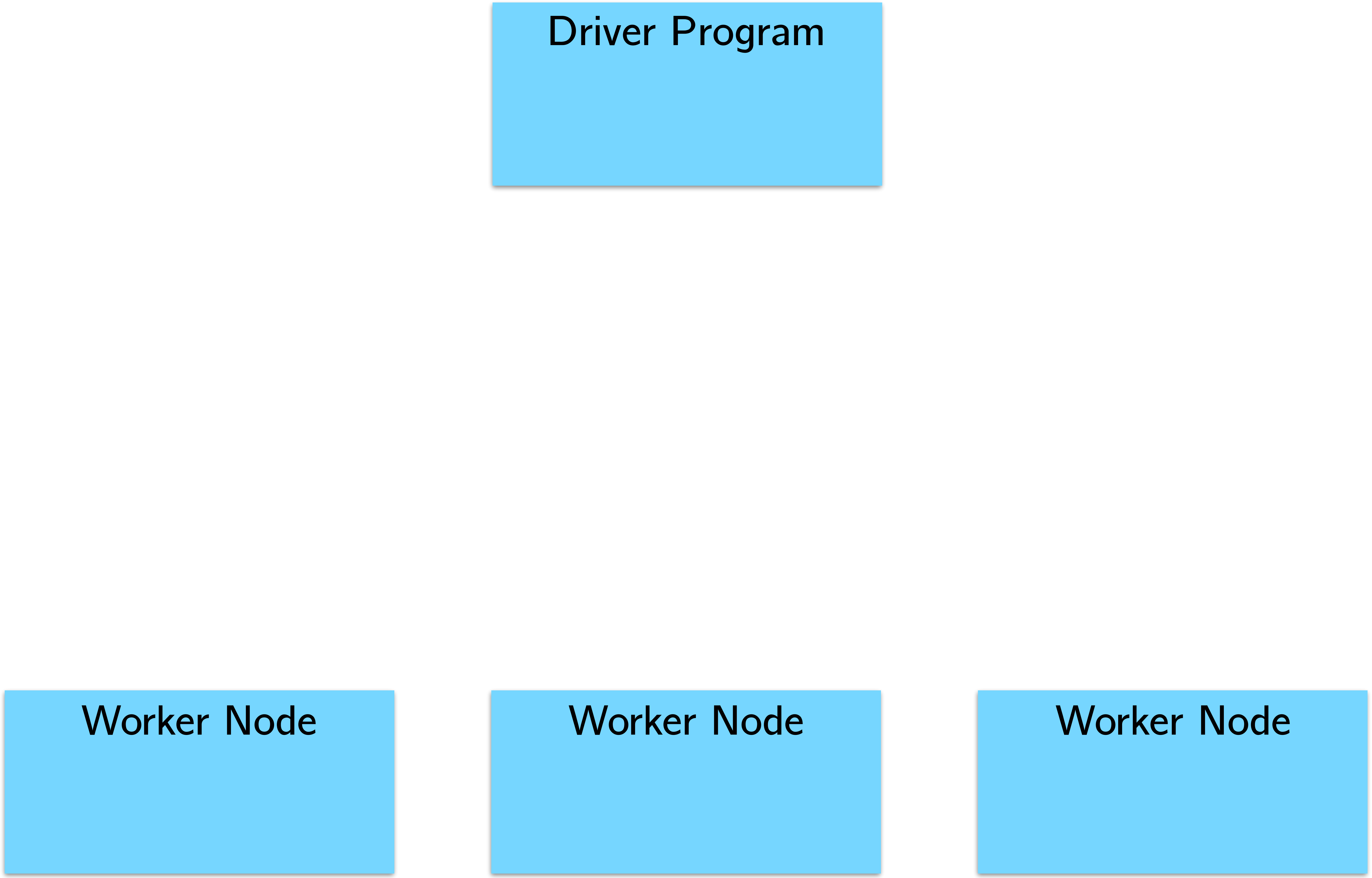
```
val people: RDD[Person] = ...
val first10 = people.take(10)
```

Where will the `Array[Person]` representing `first10` end up?

How Spark Jobs are Executed



How Spark Jobs are Executed



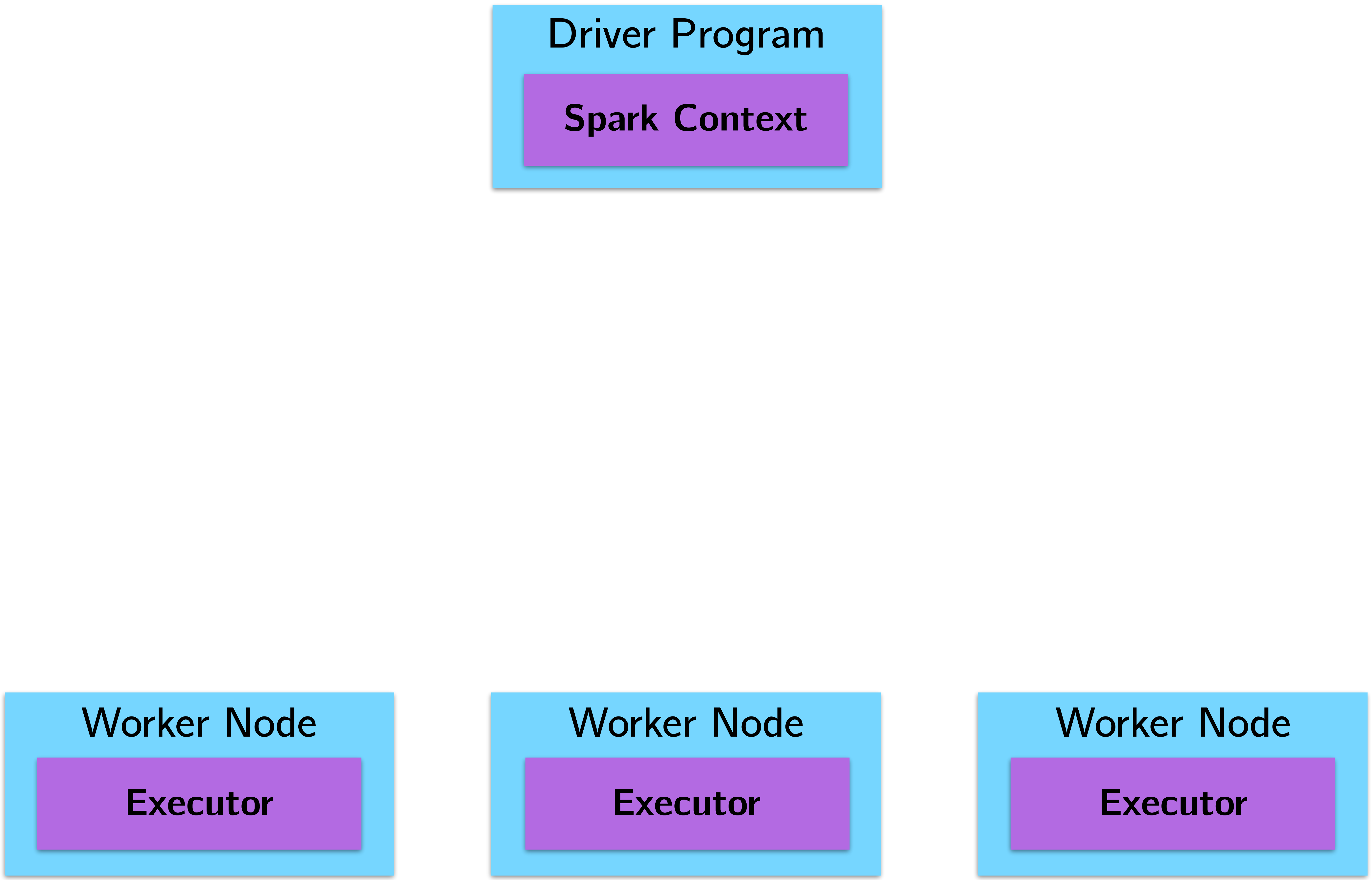
Driver Program

Worker Node

Worker Node

Worker Node

How Spark Jobs are Executed


Driver Program

Spark Context

Worker Node

Executor

Worker Node

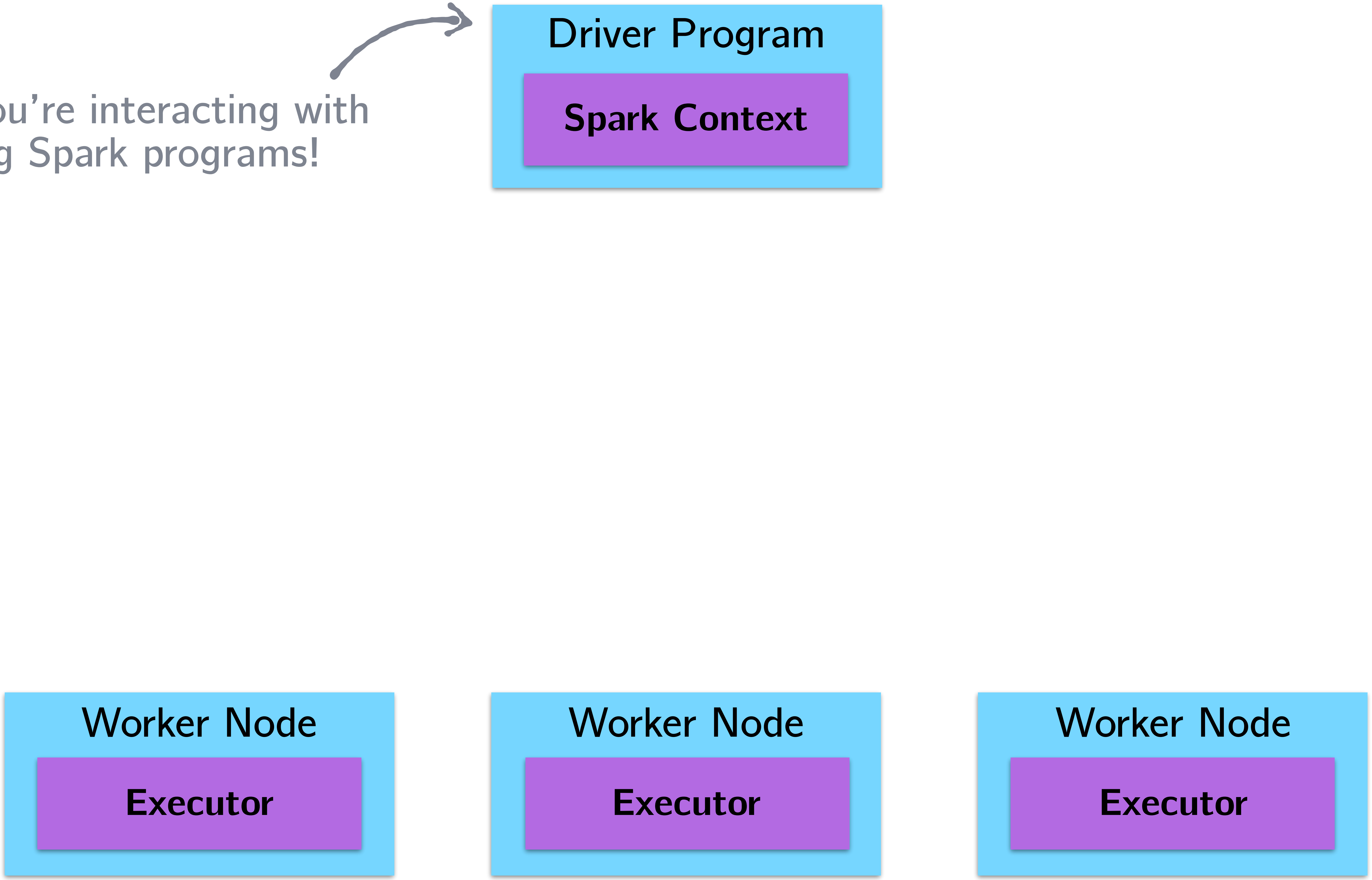
Executor

Worker Node

Executor

How Spark Jobs are Executed

This is the node you're interacting with when you're writing Spark programs!



Driver Program

Spark Context

Worker Node

Executor

Worker Node

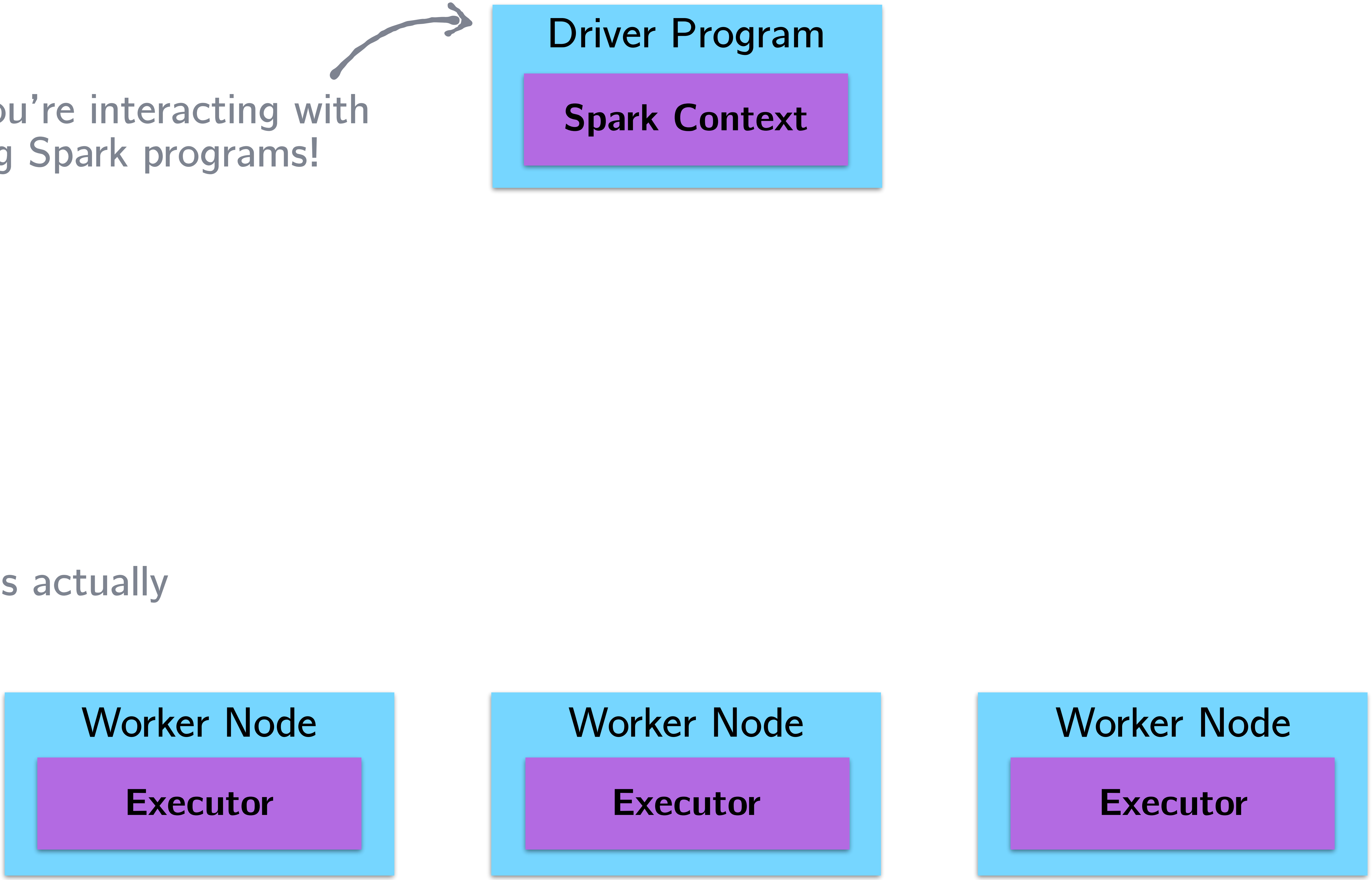
Executor

Worker Node

Executor

How Spark Jobs are Executed

This is the node you're interacting with when you're writing Spark programs!



Driver Program

Spark Context

These are the nodes actually executing the jobs!

Worker Node

Executor

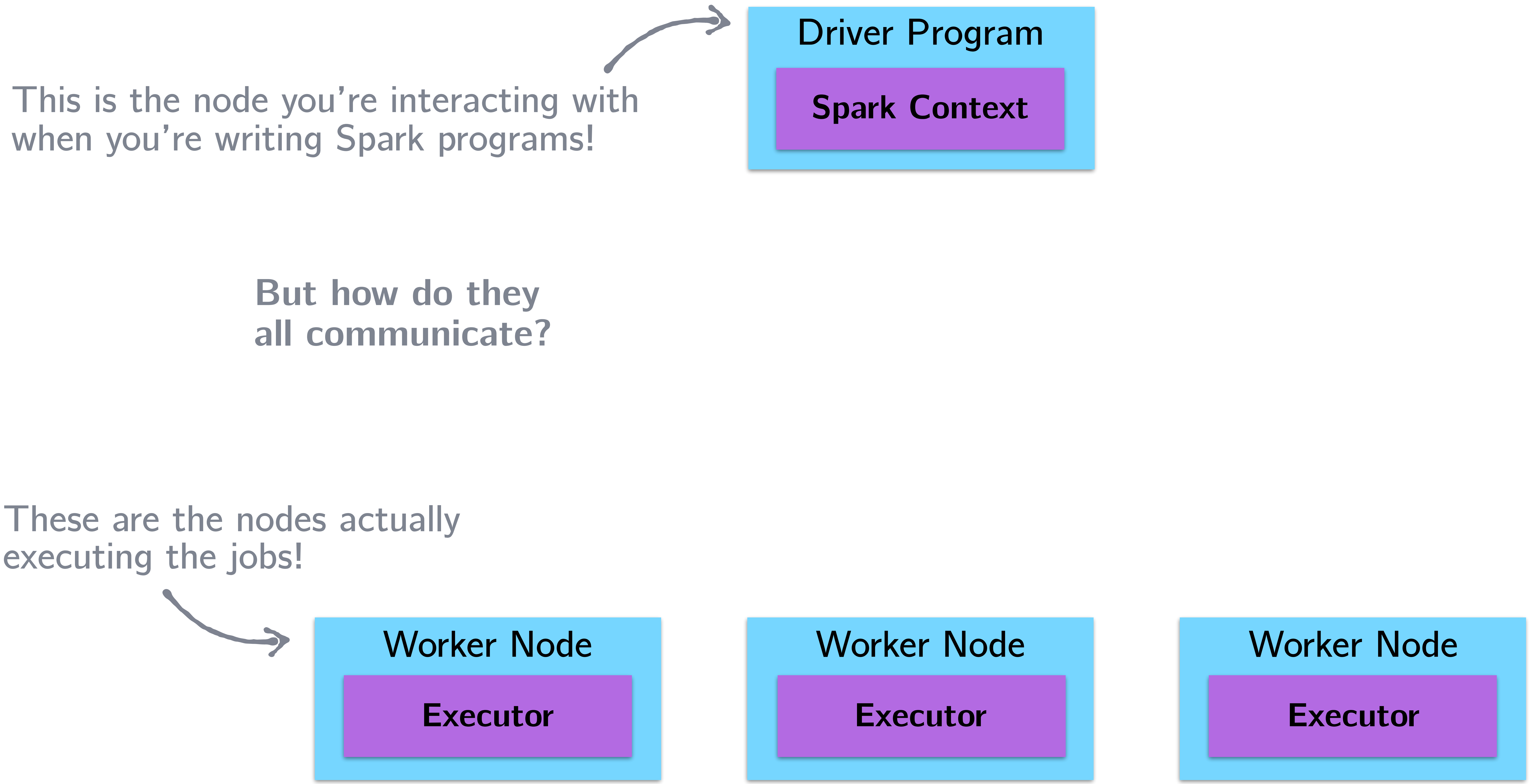
Worker Node

Executor

Worker Node

Executor

How Spark Jobs are Executed

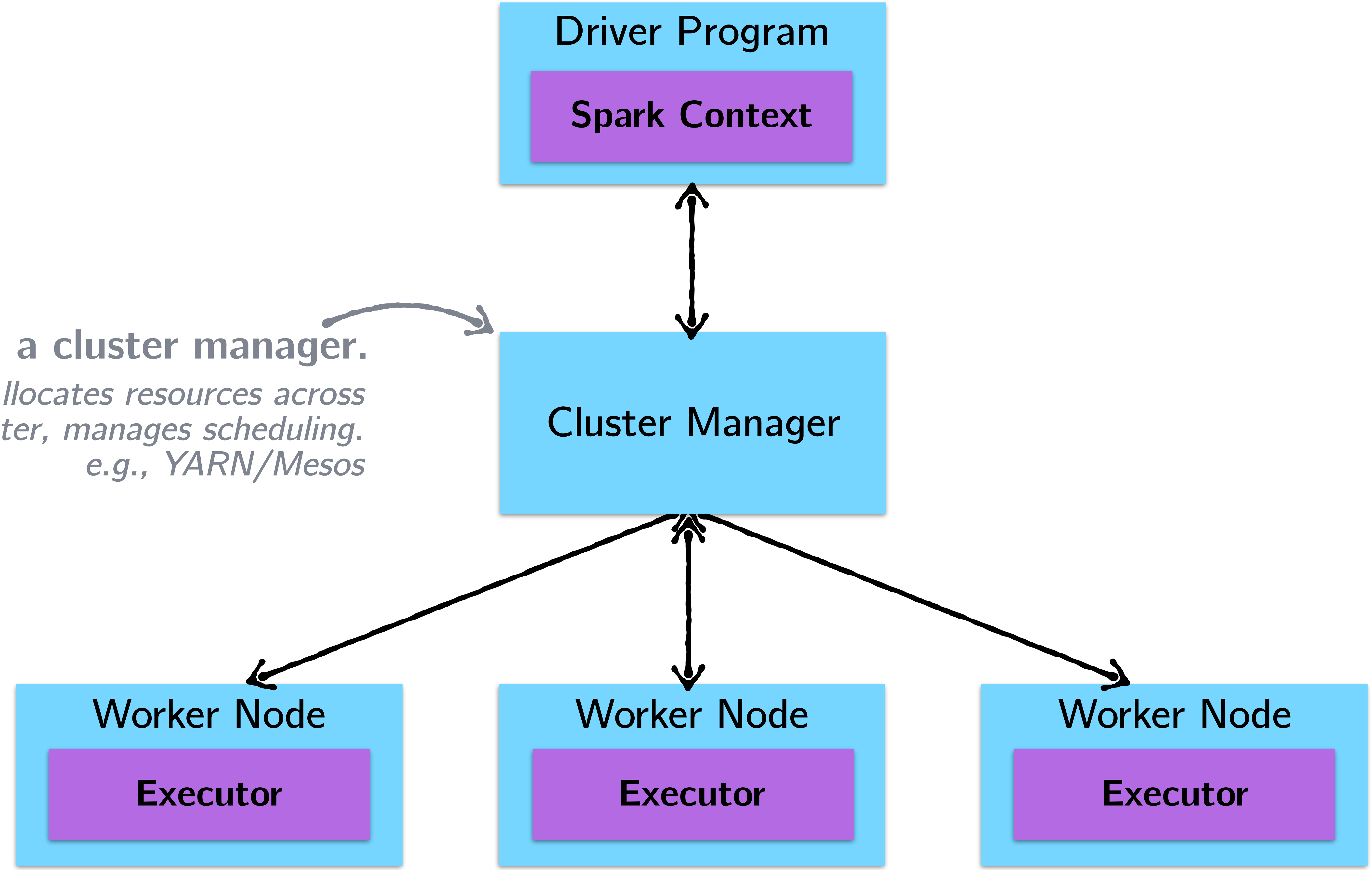


This is the node you’re interacting with when you’re writing Spark programs!

But how do they all communicate?

These are the nodes actually executing the jobs!

How Spark Jobs are Executed



Driver Program

Spark Context

Cluster Manager

Worker Node

Executor

Worker Node

Executor

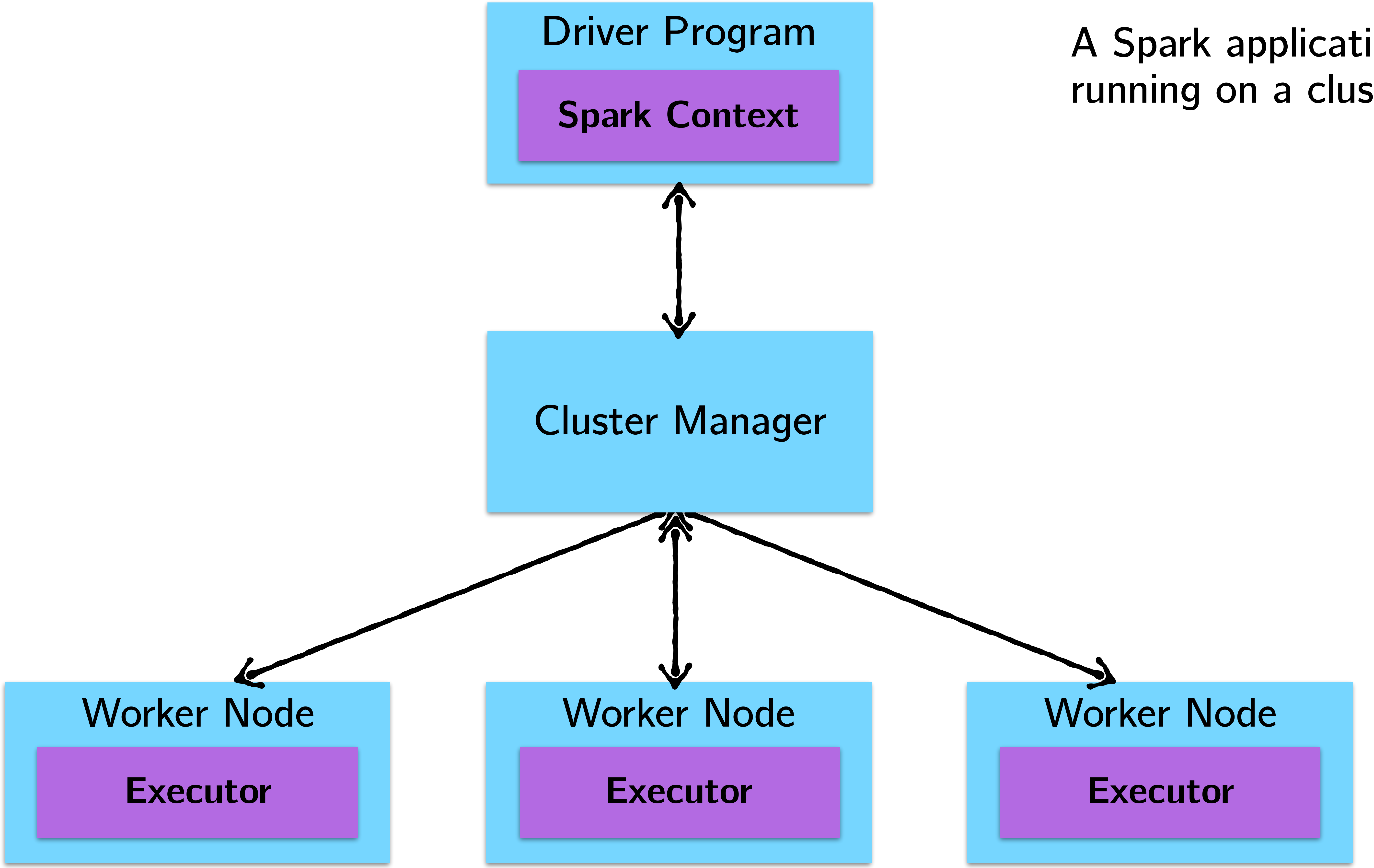
Worker Node

Executor

via a cluster manager.

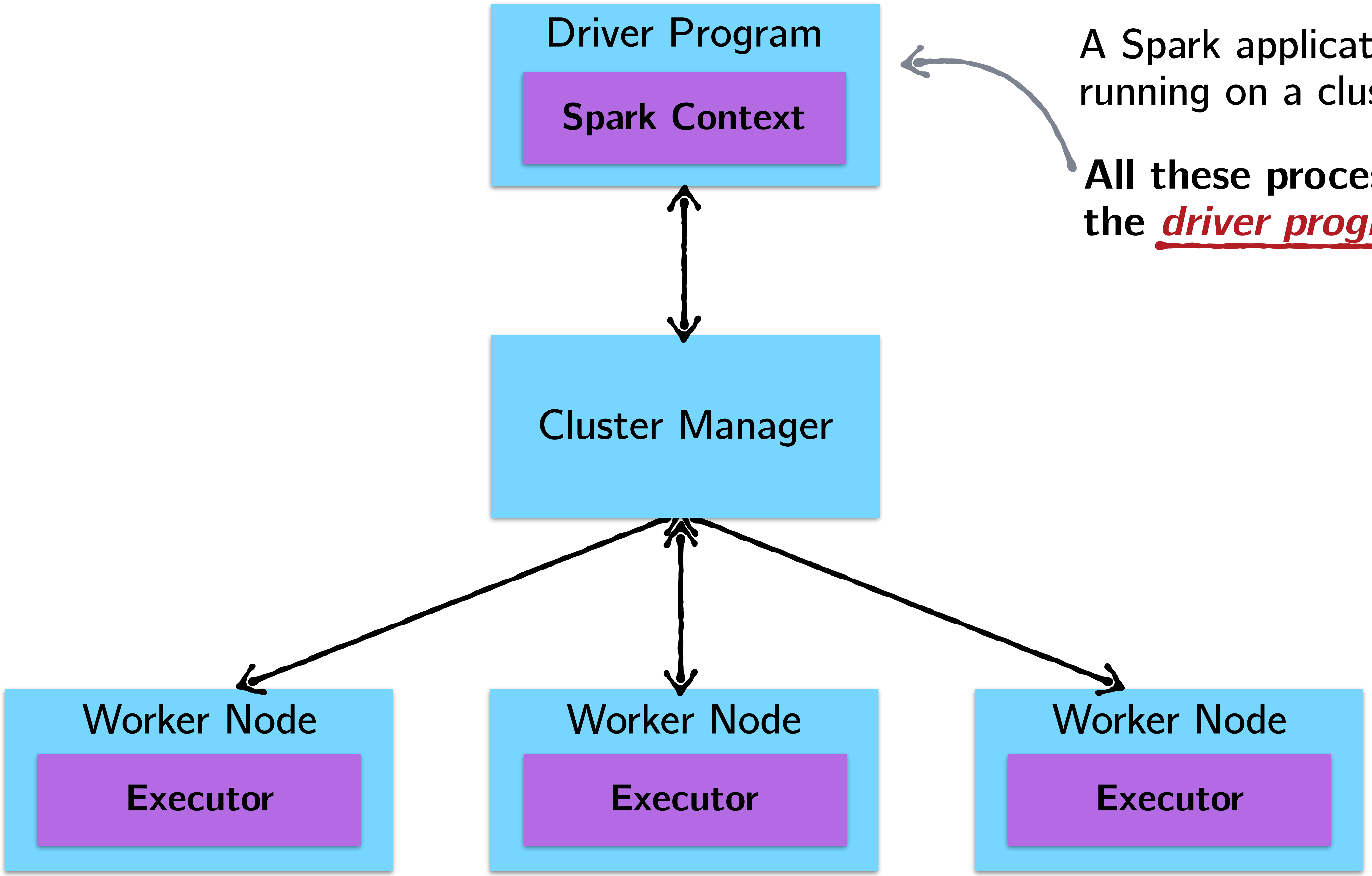
Allocates resources across cluster, manages scheduling. e.g., YARN/Mesos

How Spark Jobs are Executed



A Spark application is a set of processes running on a cluster.

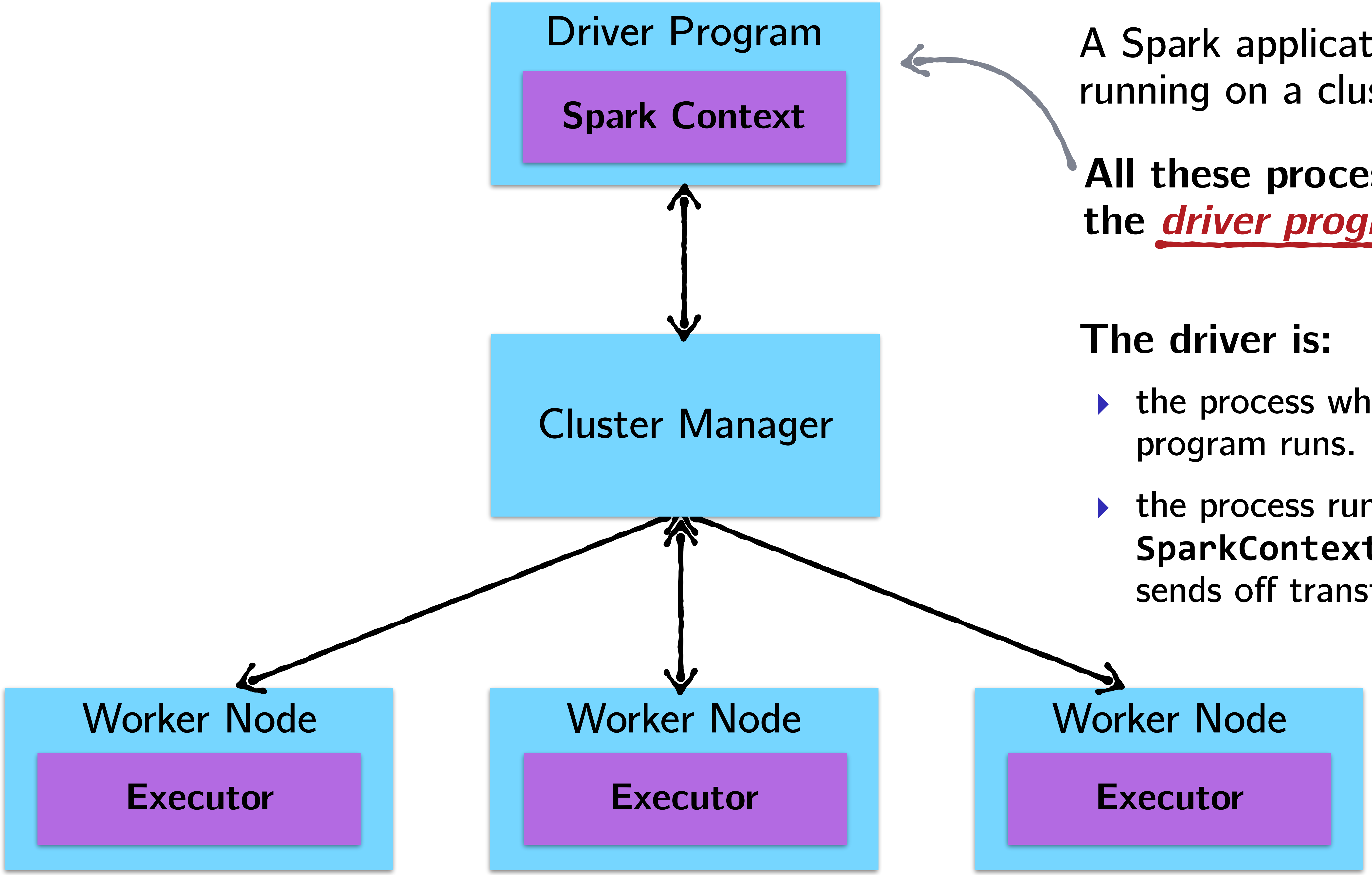
How Spark Jobs are Executed



A Spark application is a set of processes running on a cluster.

All these processes are coordinated by the *driver program*.

How Spark Jobs are Executed



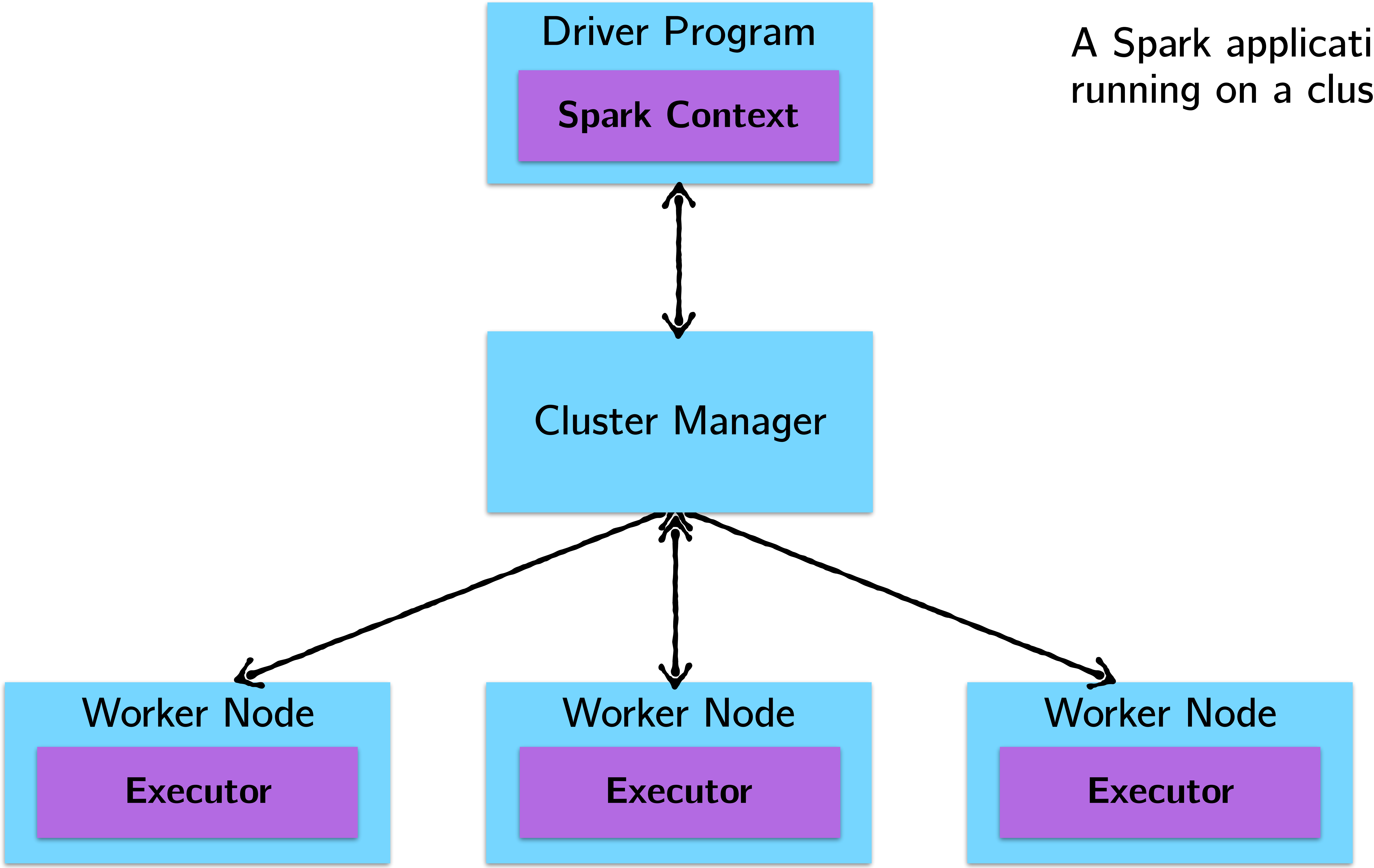
A Spark application is a set of processes running on a cluster.

All these processes are coordinated by the *driver program*.

The driver is:

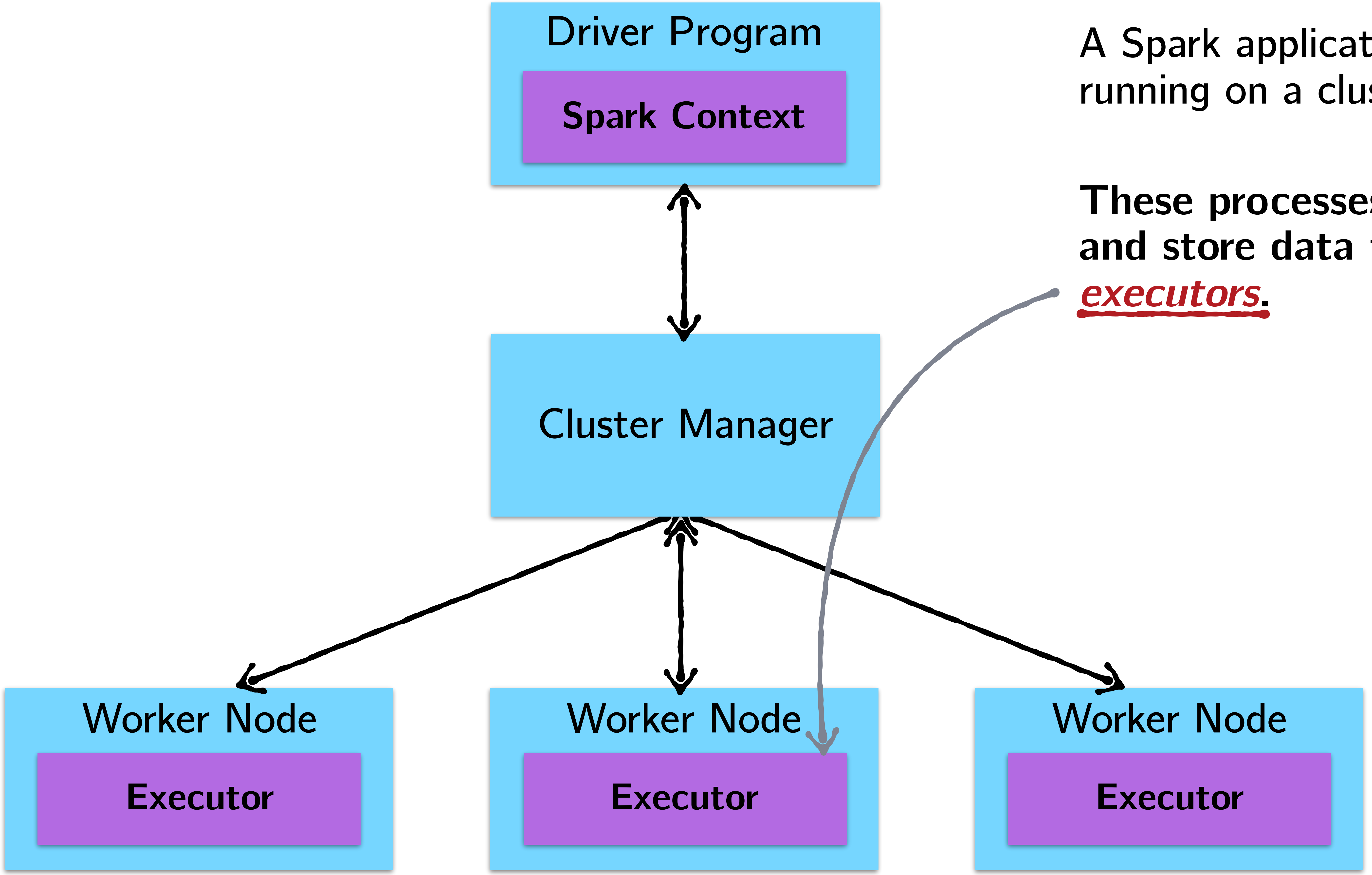
- the process where the **`main()`** method of your program runs.
- the process running the code that creates a **`SparkContext`**, creates **`RDDs`**, and stages up or sends off transformations and actions.

How Spark Jobs are Executed



A Spark application is a set of processes running on a cluster.

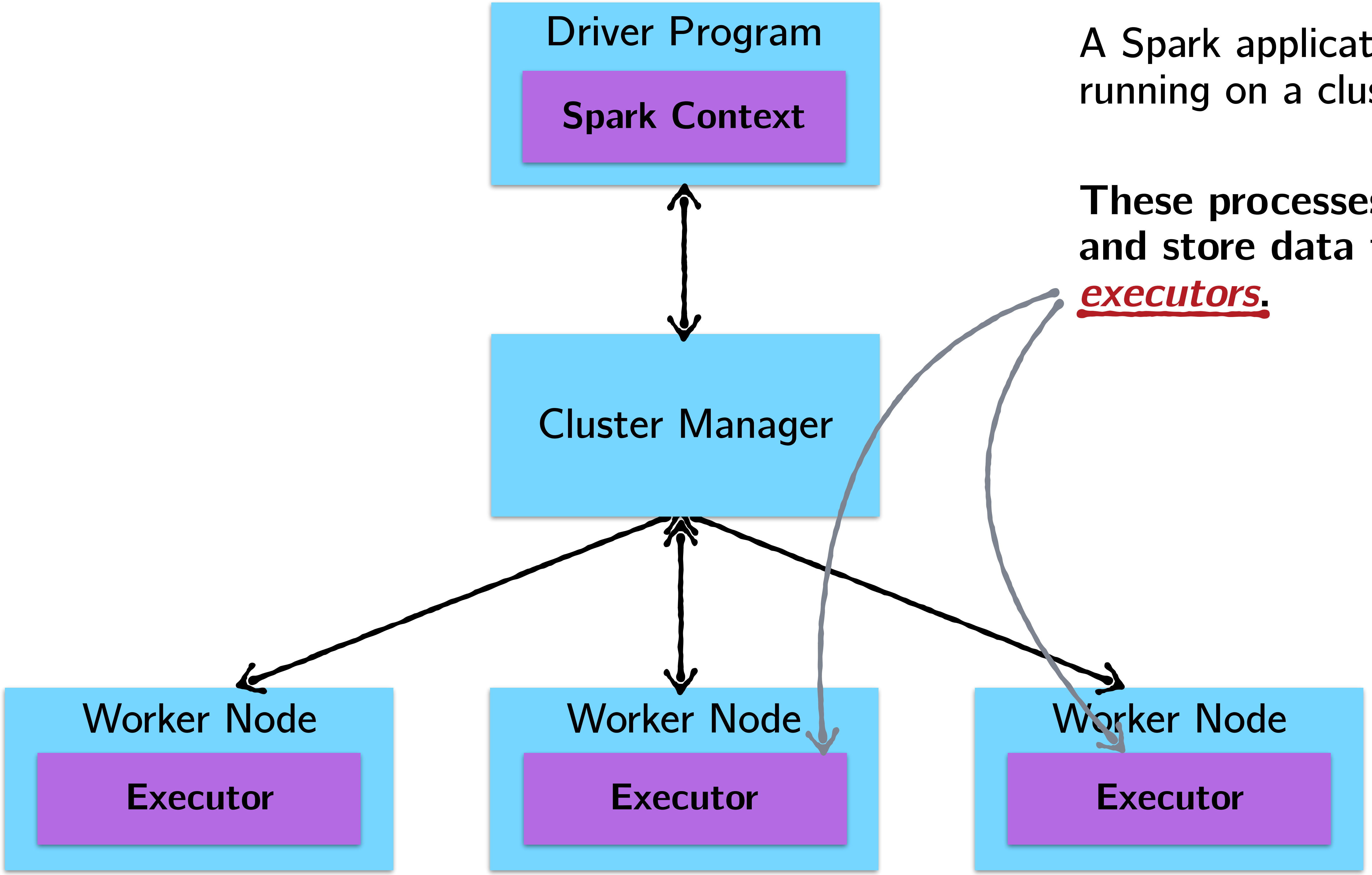
How Spark Jobs are Executed



A Spark application is a set of processes running on a cluster.

These processes that run computations and store data for your application are *executors*.

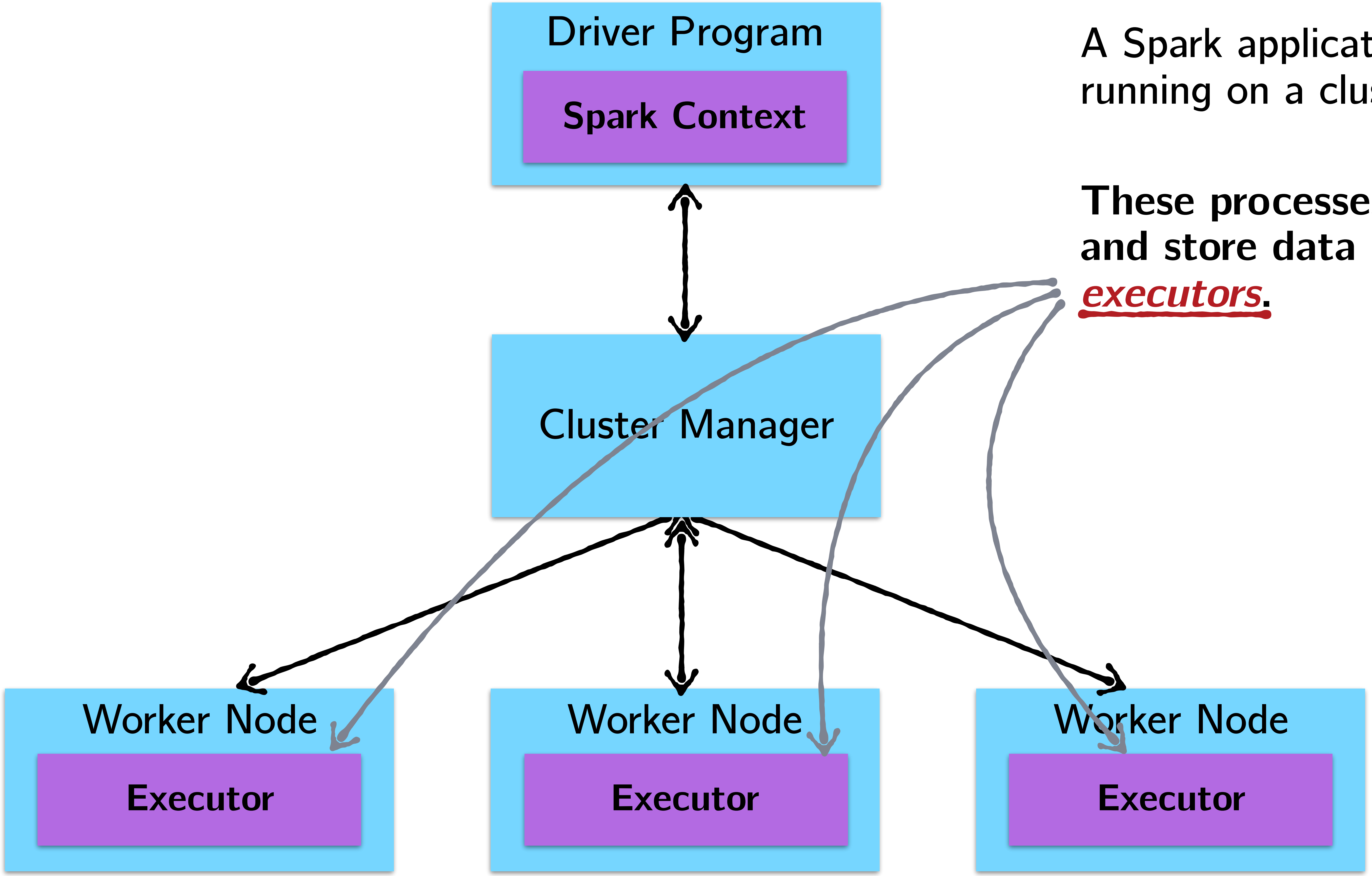
How Spark Jobs are Executed



A Spark application is a set of processes running on a cluster.

These processes that run computations and store data for your application are *executors*.

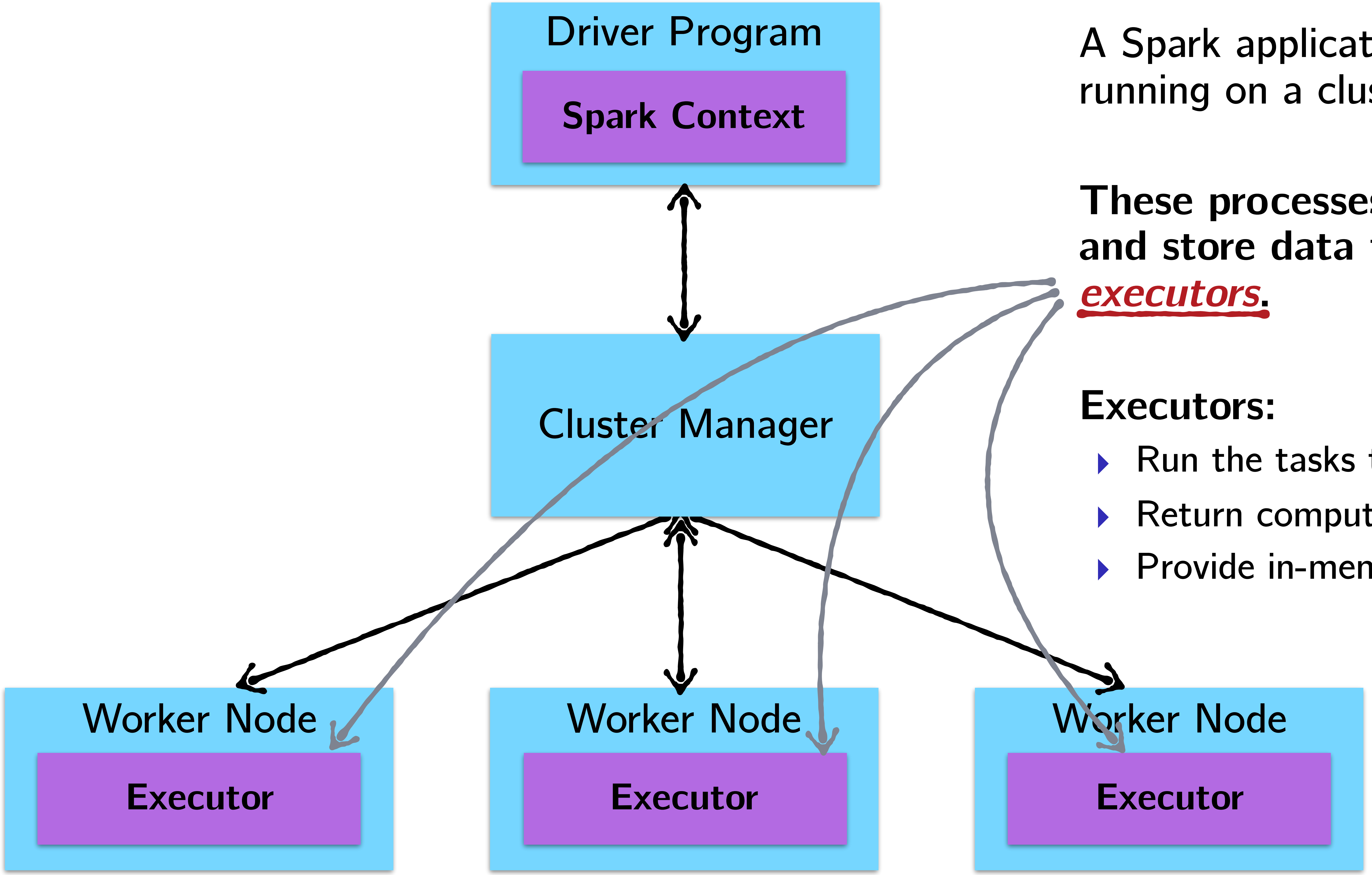
How Spark Jobs are Executed



A Spark application is a set of processes running on a cluster.

These processes that run computations and store data for your application are *executors*.

How Spark Jobs are Executed



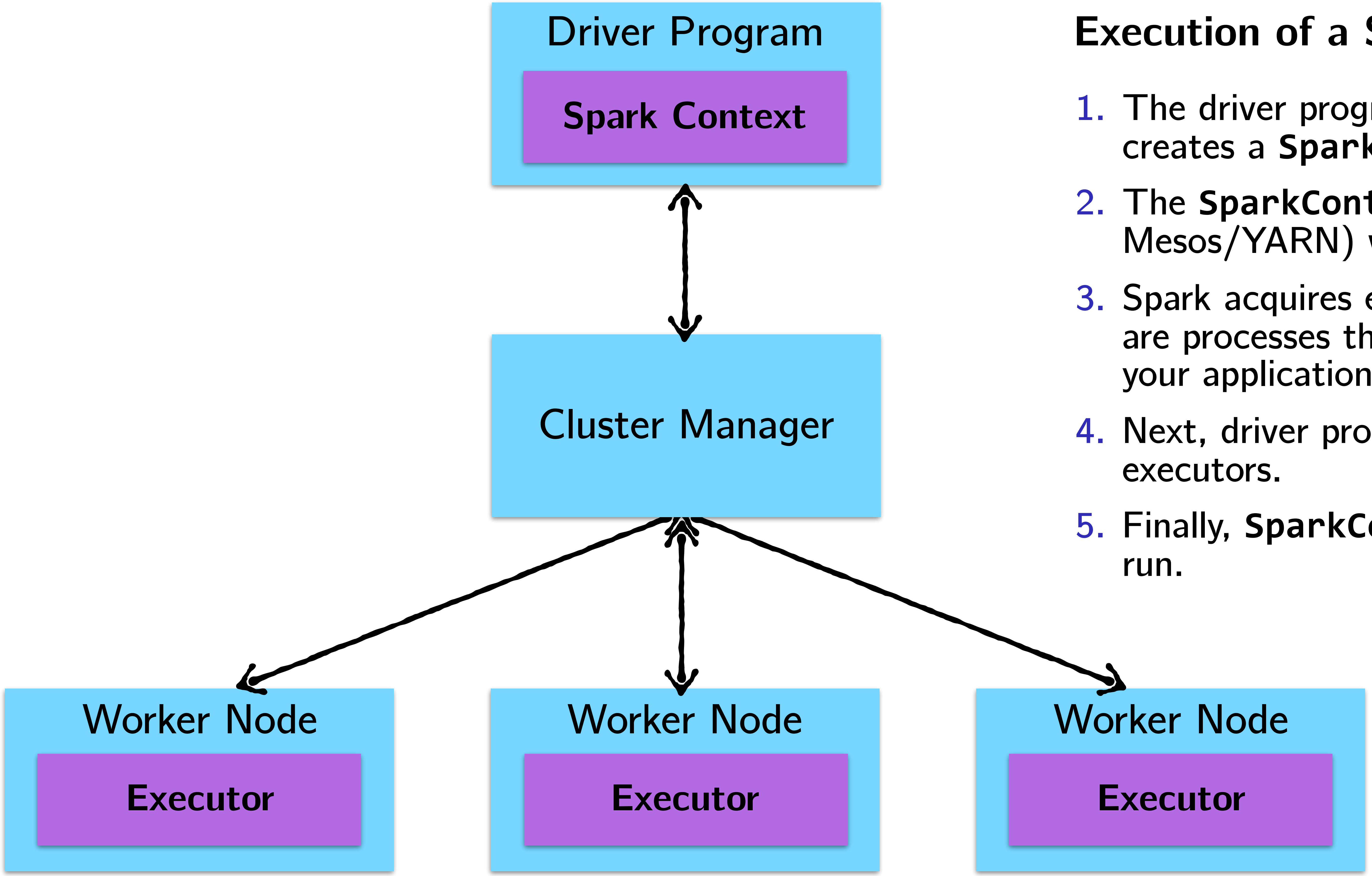
A Spark application is a set of processes running on a cluster.

These processes that run computations and store data for your application are *executors*.

Executors:

- Run the tasks that represent the application.
- Return computed results to the driver.
- Provide in-memory storage for cached RDDs.

How Spark Jobs are Executed



Execution of a Spark program:

1. The driver program runs the Spark application, which creates a **`SparkContext`** upon start-up.
2. The **`SparkContext`** connects to a cluster manager (e.g., Mesos/YARN) which allocates resources.
3. Spark acquires executors on nodes in the cluster, which are processes that run computations and store data for your application.
4. Next, driver program sends your application code to the executors.
5. Finally, **`SparkContext`** sends tasks for the executors to run.

Back to Example 1: A Simple println

Let's start with an example. Assume we have an RDD populated with Person objects:

```
case class Person(name: String, age: Int)
```

What does the following code snippet do?

```
val people: RDD[Person] = ...
people.foreach(println)
```

Back to Example 1: A Simple `println`

Let's start with an example. Assume we have an RDD populated with `Person` objects:

```
case class Person(name: String, age: Int)
```

What does the following code snippet do?

```
val people: RDD[Person] = ...
people.foreach(println)
```

On the driver: Nothing. Why?

Back to Example 1: A Simple println

Let's start with an example. Assume we have an RDD populated with Person objects:

```
case class Person(name: String, age: Int)
```

What does the following code snippet do?

```
val people: RDD[Person] = ...
people.foreach(println)
```

On the driver: Nothing. Why?

Recall that foreach is an action, with return type Unit. Therefore, it is eagerly executed on the executors, not the driver. Therefore, any calls to println are happening on the stdout of worker nodes and are thus not visible in the stdout of the driver node.

Back to Example 2: A Simple take

What about here? Assume we have an RDD populated with the same definition of Person objects:

```scala
case class Person(name: String, age: Int)
```

What does the following code snippet do?

```scala
val people: RDD[Person] = ...
val first10 = people.take(10)
```

Where will the Array[Person] representing first10 end up?

Back to Example 2: A Simple take

What about here? Assume we have an RDD populated with the same definition of Person objects:

```
case class Person(name: String, age: Int)
```

What does the following code snippet do?

```
val people: RDD[Person] = ...
val first10 = people.take(10)
```

Where will the Array[Person] representing first10 end up?

The driver program.

In general, executing an action involves communication between worker nodes and the node running the driver program.

Cluster Toplogy Matters!

Moral of the story:

To make effective use of RDDs, you have to understand a little bit about how Spark works under the hood.

Due an API which is mixed eager/lazy, it's not always immediately obvious upon first glance on what part of the cluster a line of code might run on. **It's on you to know where your code is executing!**

Even though RDDs look like regular Scala collections upon first glance, unlike collections, RDDs require you to have a good grasp of the underlying infrastructure they are running on.