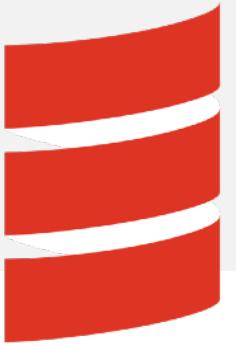
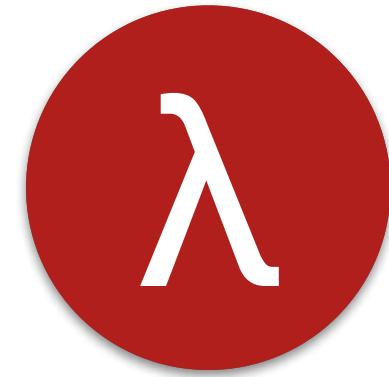


So far in the Scala courses...



Focused on:

- ▶ **Basics of Functional Programming**. Slowly building up on fundamentals.
- ▶ **Parallelism**. Experience with underlying execution in shared memory parallelism.

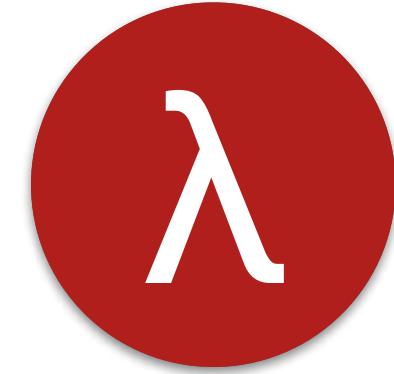


So far in the Scala courses...



Focused on:

- ▶ **Basics of Functional Programming**. Slowly building up on fundamentals.
- ▶ **Parallelism**. Experience with underlying execution in shared memory parallelism.



This course:

Not a machine learning or data science course!

- ▶ This is a course about distributed data parallelism in Spark.
- ▶ Extending familiar functional abstractions like functional lists over large clusters.
- ▶ Context: analyzing large data sets.

Why Scala? Why Spark?

Why Scala? Why Spark?

Normally:

Data science and analytics is done “*in the small*”, in R/Python/MATLAB, etc

Why Scala? Why Spark?

Normally:

Data science and analytics is done “*in the small*”, in R/Python/MATLAB, etc

If your dataset ever gets too large to fit into memory,
these languages/frameworks won’t allow you to scale. You’ve to reimplement
everything in some other language or system.

Why Scala? Why Spark?

Normally:

Data science and analytics is done “*in the small*”, in R/Python/MATLAB, etc

If your dataset ever gets too large to fit into memory,
these languages/frameworks won’t allow you to scale. You’ve to reimplement
everything in some other language or system.

Oh yeah, there’s also the massive shift in industry to data-oriented decision making too!

...and many applications are “data science in the large”.

Why Scala? Why Spark?

By using a language like Scala, it's easier to scale your small problem to the large with Spark, whose API is almost 1-to-1 with Scala's collections.

That is, by working in Scala, in a functional style, you can quickly scale your problem from one node to tens, hundreds, or even thousands by leveraging Spark, successful and performant large-scale data processing framework which looks a lot like Scala Collections!



Why Spark?

Spark is...

- ▶ **More expressive.** APIs modeled after Scala collections. Look like functional lists! Richer, more composable operations possible than in MapReduce.



Why Spark?

Spark is...

- ▶ **More expressive.** APIs modeled after Scala collections. Look like functional lists! Richer, more composable operations possible than in MapReduce.
- ▶ **Performant.** Not only performant in terms of running time... But also in terms of developer productivity!
Interactive!



Why Spark?

Spark is...

- ▶ **More expressive.** APIs modeled after Scala collections. Look like functional lists! Richer, more composable operations possible than in MapReduce.
- ▶ **Performant.** Not only performant in terms of running time... But also in terms of developer productivity! Interactive!
- ▶ **Good for data science.** Not just because of performance, but because it enables *iteration*, which is required by most algorithms in a data scientist's toolbox.



Also good to know...

Spark and Scala skills are in extremely high demand!

In this course you'll learn...

- ▶ Extending data parallel paradigm to the distributed case, using Spark.
- ▶ Spark's programming model
- ▶ Distributing computation, and cluster topology in Spark
- ▶ How to improve performance; data locality, how to avoid recomputation and shuffles in Spark.
- ▶ Relational operations with DataFrames and Datasets

Prerequisites

Builds on the material taught in the previous Scala courses.

- ▶ **Principles of Functional Programming in Scala.**
- ▶ **Functional Program Design in Scala**
- ▶ **Parallel Programming (in Scala)**

Or at minimum, some familiarity with Scala.

Books, Resources

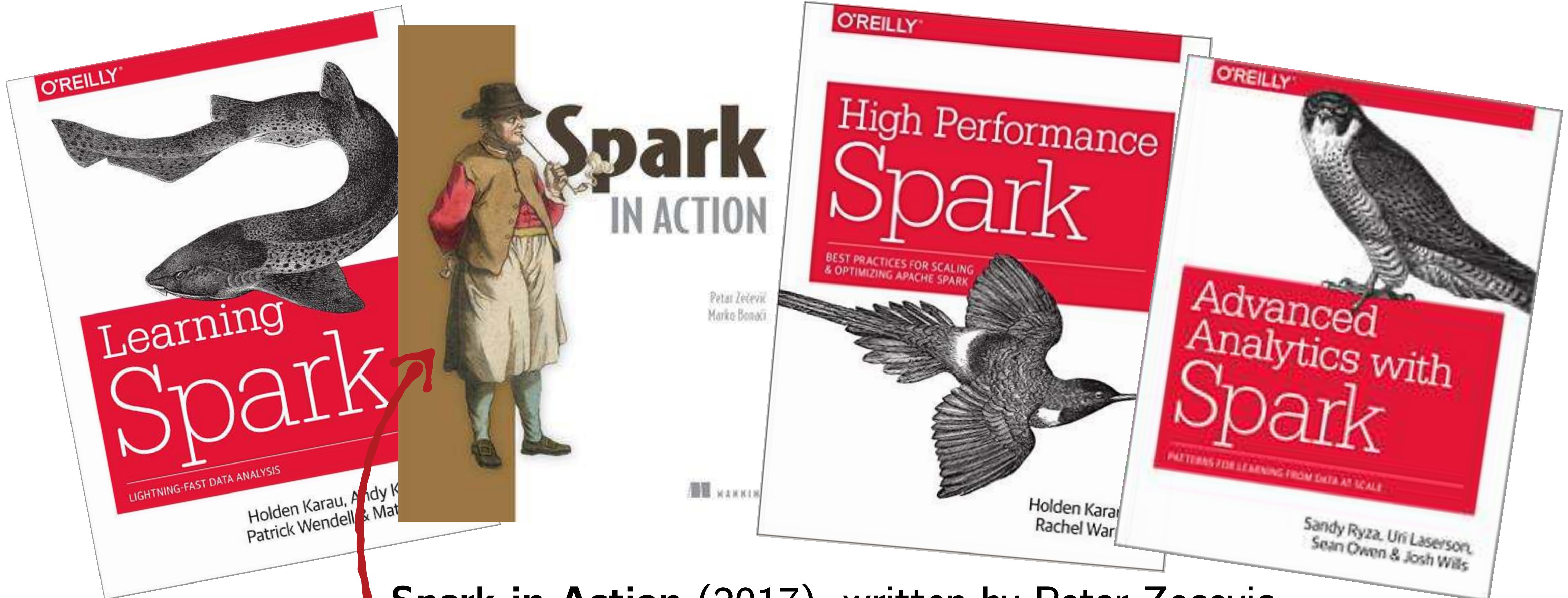
Many excellent books released in the past year or two!



Learning Spark (2015), written by Holden Karau,
Andy Konwinski, Patrick Wendell, and Matei Zaharia

Books, Resources

Many excellent books released in the past year or two!



Spark in Action (2017), written by Petar Zecevic
and Marko Bonaci

Books, Resources

Many excellent books released in the past year or two!



High Performance Spark (in progress), written by
Holden Karau and Rachel Warren

Books, Resources

Many excellent books released in the past year or two!



Advanced Analytics with Spark (2015), written by
Sandy Ryza, Uri Laserson, Sean Owen, and Josh Wills

Books, Resources



WE ARE HIRING!

Pricing

Explore

About

Blog

Sign In

Sign Up

jaceklaskowski > Mastering Apache Spark 2

Updated an hour ago

ABOUT

138 DISCUSSIONS

0 CHANGE REQUESTS

Mastering Apache Spark 2, by Jacek Laskowski

★ Star

682

Subscribe

308

Download PDF

Read

Mastering Apache Spark 2

Welcome to Mastering Apache Spark 2 (aka #SparkLikePro)!

I'm [Jacek Laskowski](#), an **independent consultant** who is passionate about **Apache Spark**, Apache Kafka, Scala, sbt (with some flavour of Apache Mesos, Hadoop YARN, and DC/OS). I lead [Warsaw Scala Enthusiasts](#) and [Warsaw Spark](#) meetups in Warsaw, Poland.

Tools

As in all other Scala courses...

- ▶ IDE of your choice
- ▶ sbt
- ▶ Databricks Community Edition (*optional*)

Free hosted in-browser Spark notebook. Spark “cluster” managed by Databricks so you don’t have to worry about it. 6GB of memory for you to experiment with.

Assignments

Like all other Scala courses, this course comes with autograders!

**Course features 3 auto-graded assignments that require
you to do analyses on real-life datasets.**



ÉCOLE POLYTECHNIQUE
FÉDÉRALE DE LAUSANNE

Data-Parallel to Distributed Data-Parallel

Big Data Analysis with Scala and Spark

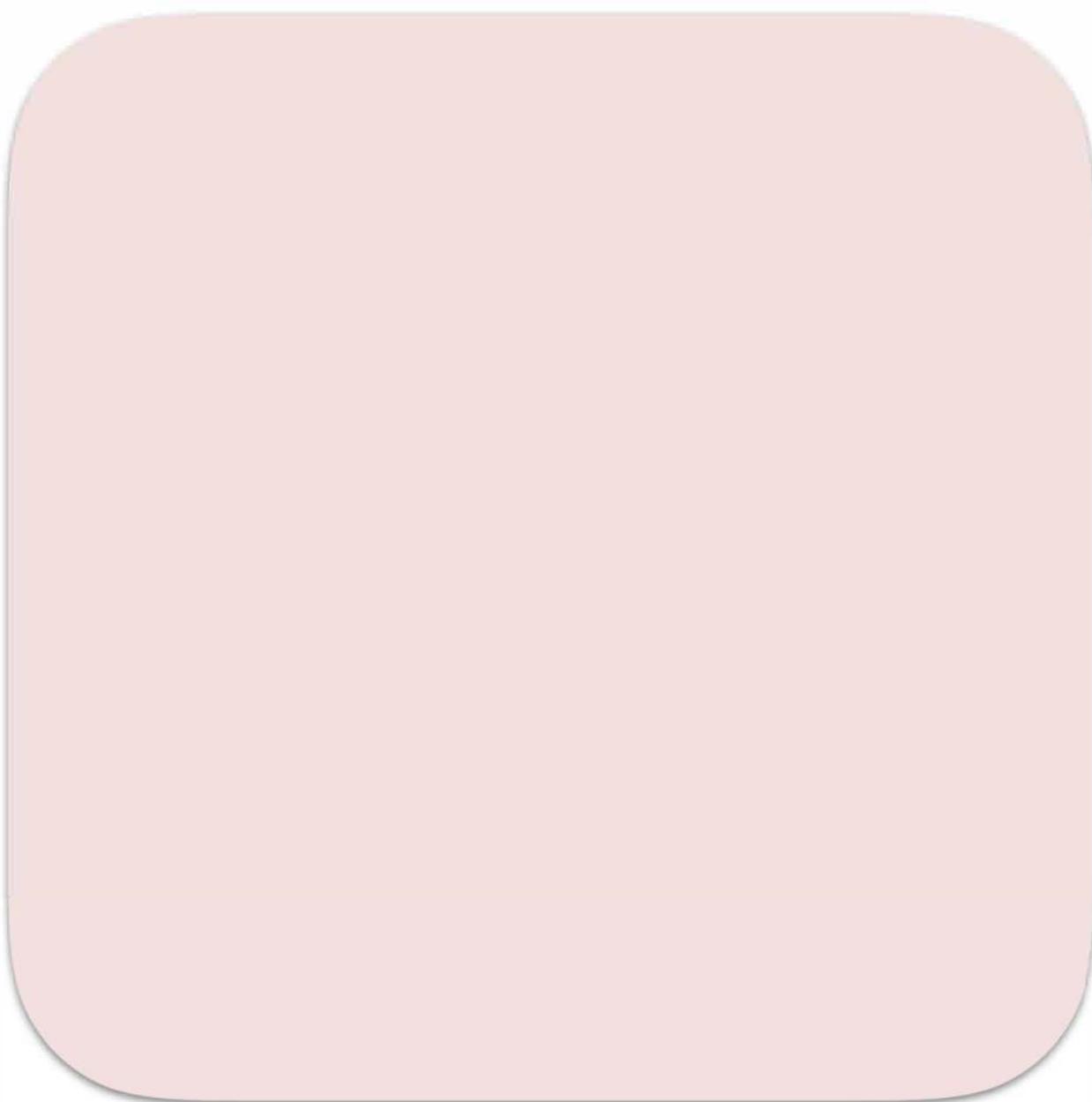
Heather Miller

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?



Compute Node
(Shared Memory)

Visualizing Shared Memory Data Parallelism

What does data-parallel look like?

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



Collection

Compute Node
(Shared Memory)

Visualizing Shared Memory Data Parallelism

What does data-parallel look like?



Compute Node
(Shared Memory)

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

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?



Compute Node
(Shared Memory)

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

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?



Compute Node
(Shared Memory)

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

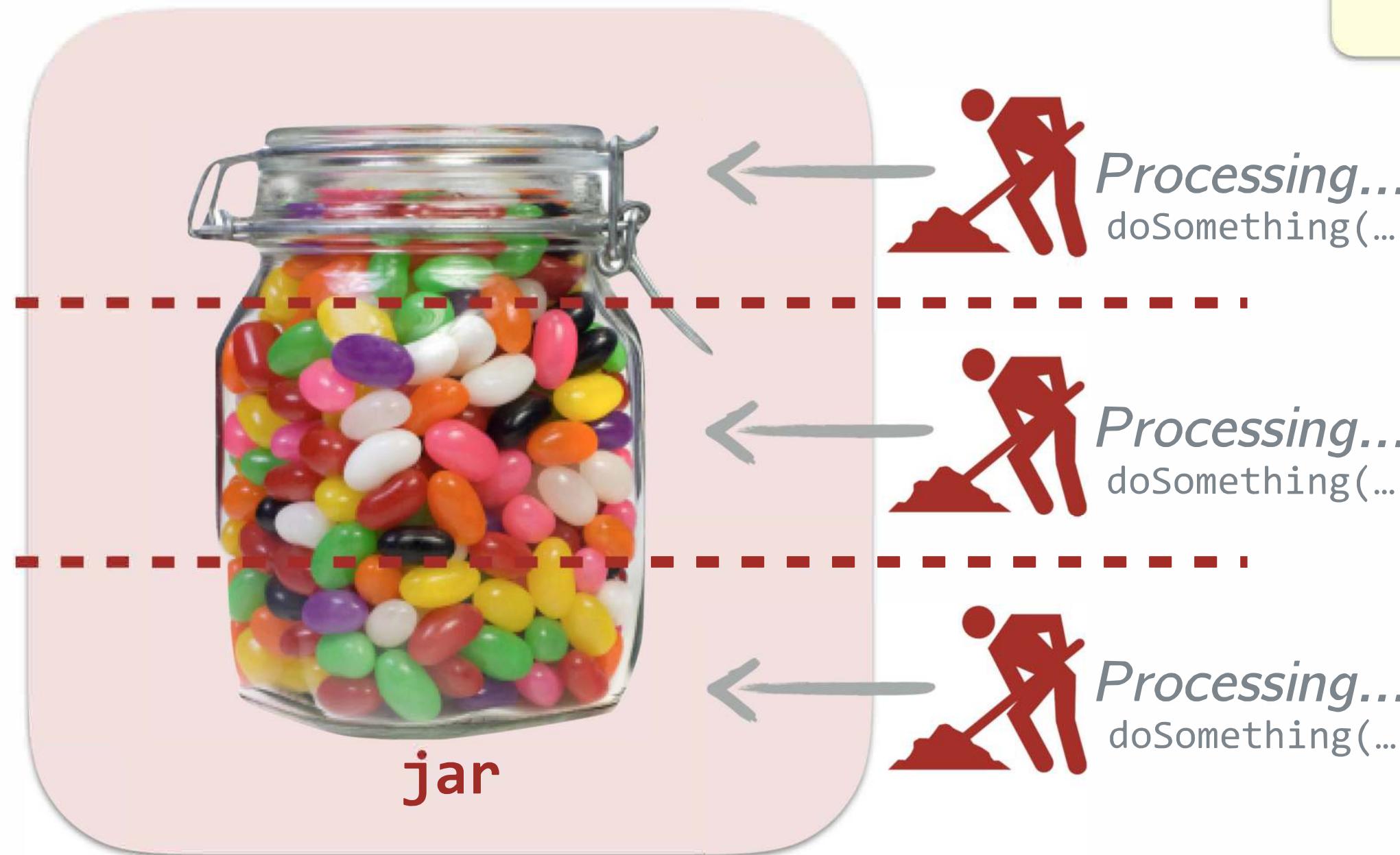
Shared memory data parallelism:

- ▶ 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))
```



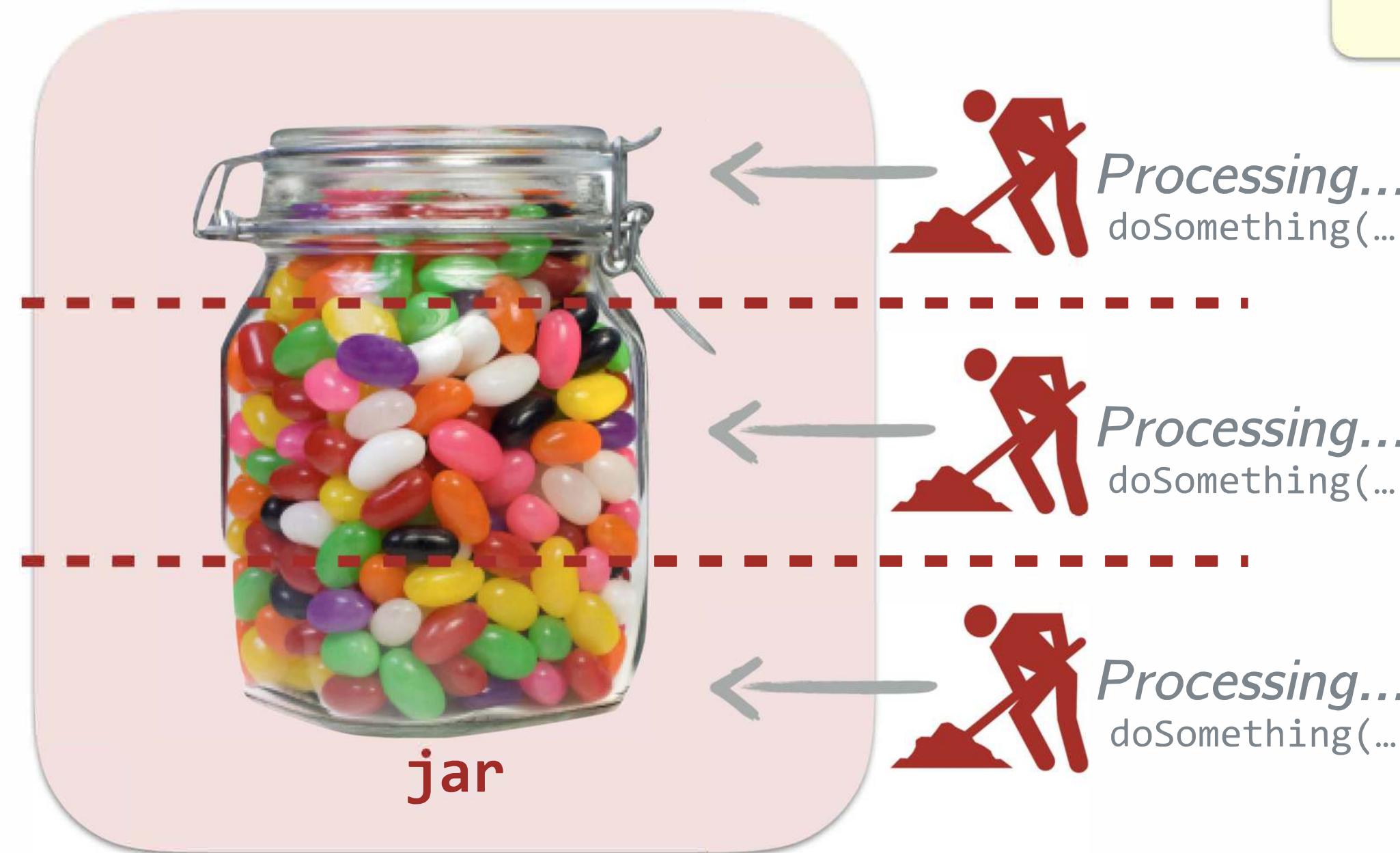
Compute Node
(Shared Memory)

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?



Compute Node
(Shared Memory)

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

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?

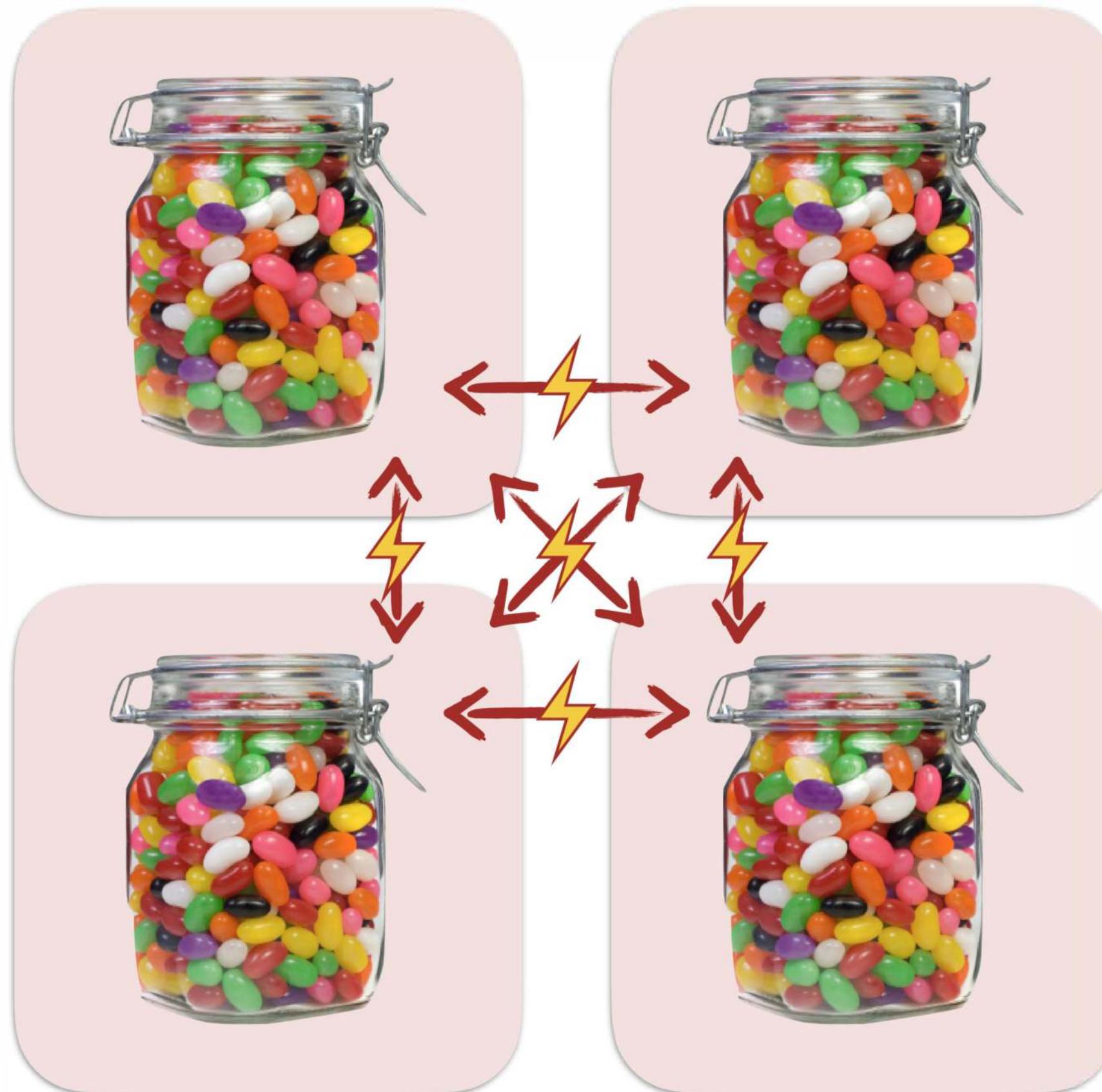


Distributed data parallelism:

- ▶ 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:

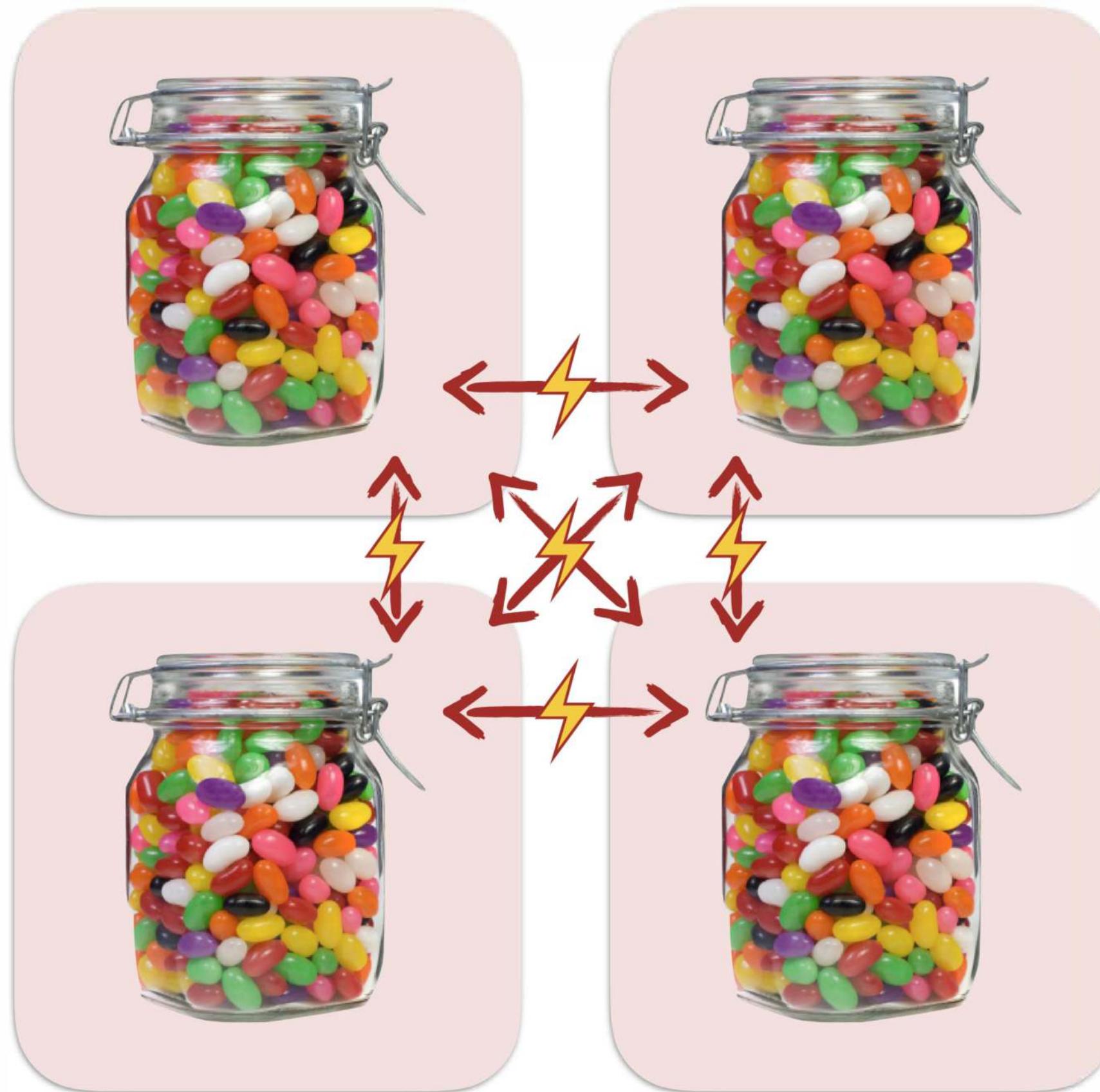
- ▶ Split the data **over several nodes**.
- ▶ **Nodes** independently operate on the data shards in parallel.
- ▶ Combine when done (if necessary).

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)**

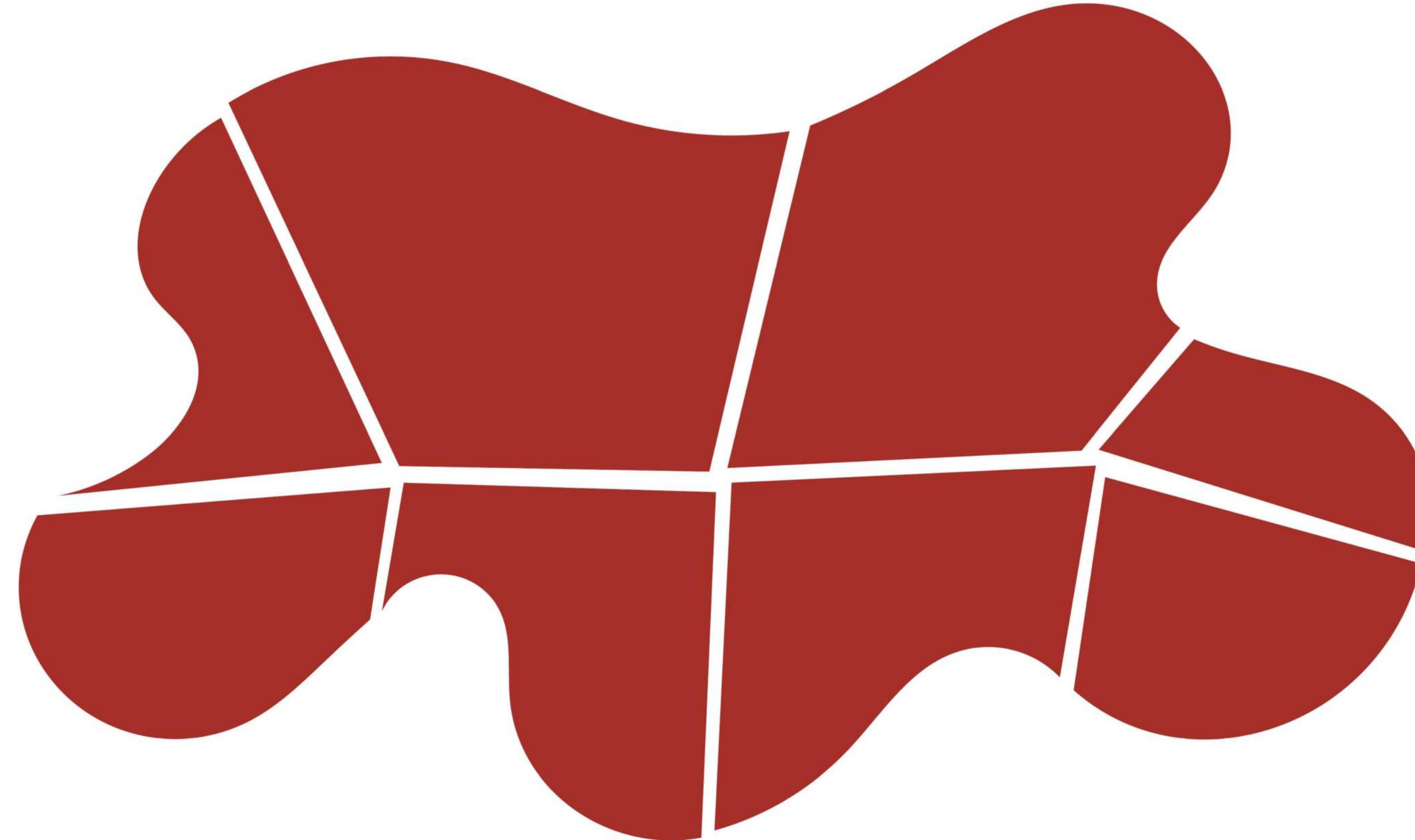
Distributed Data-Parallel: High Level Illustration



wikipedia
reduced, 48.4GB

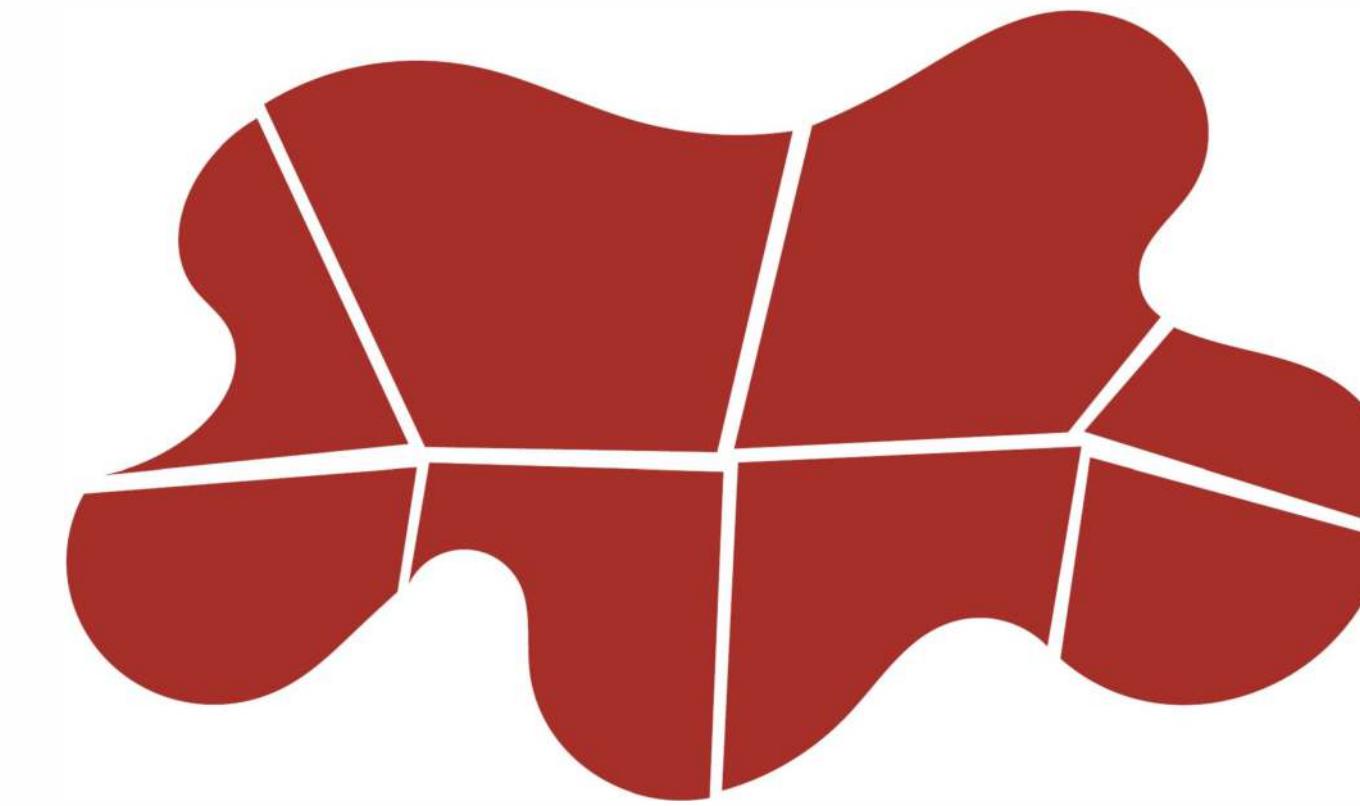
Given some large dataset that can't fit into memory on a single node...

Distributed Data-Parallel: High Level Illustration



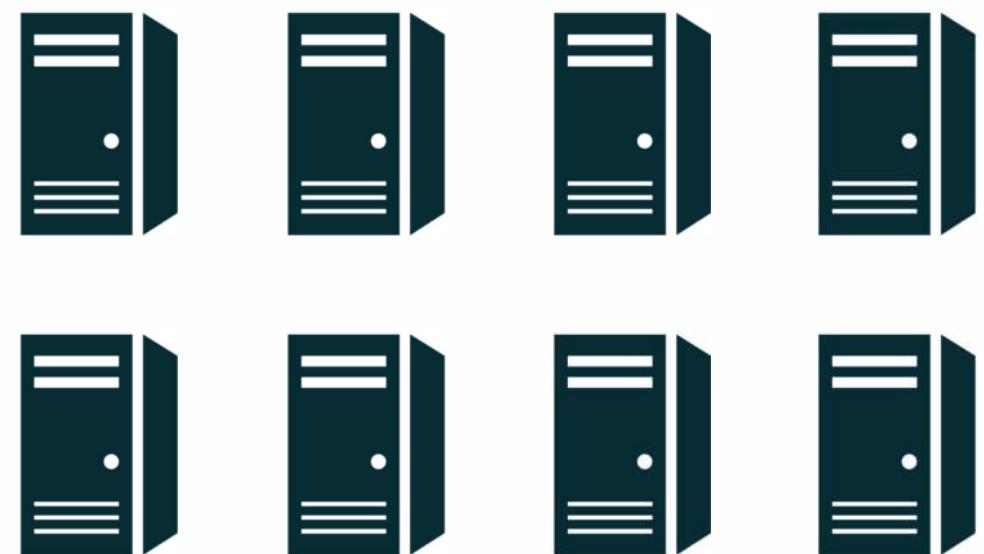
Chunk up the data...

Distributed Data-Parallel: High Level Illustration

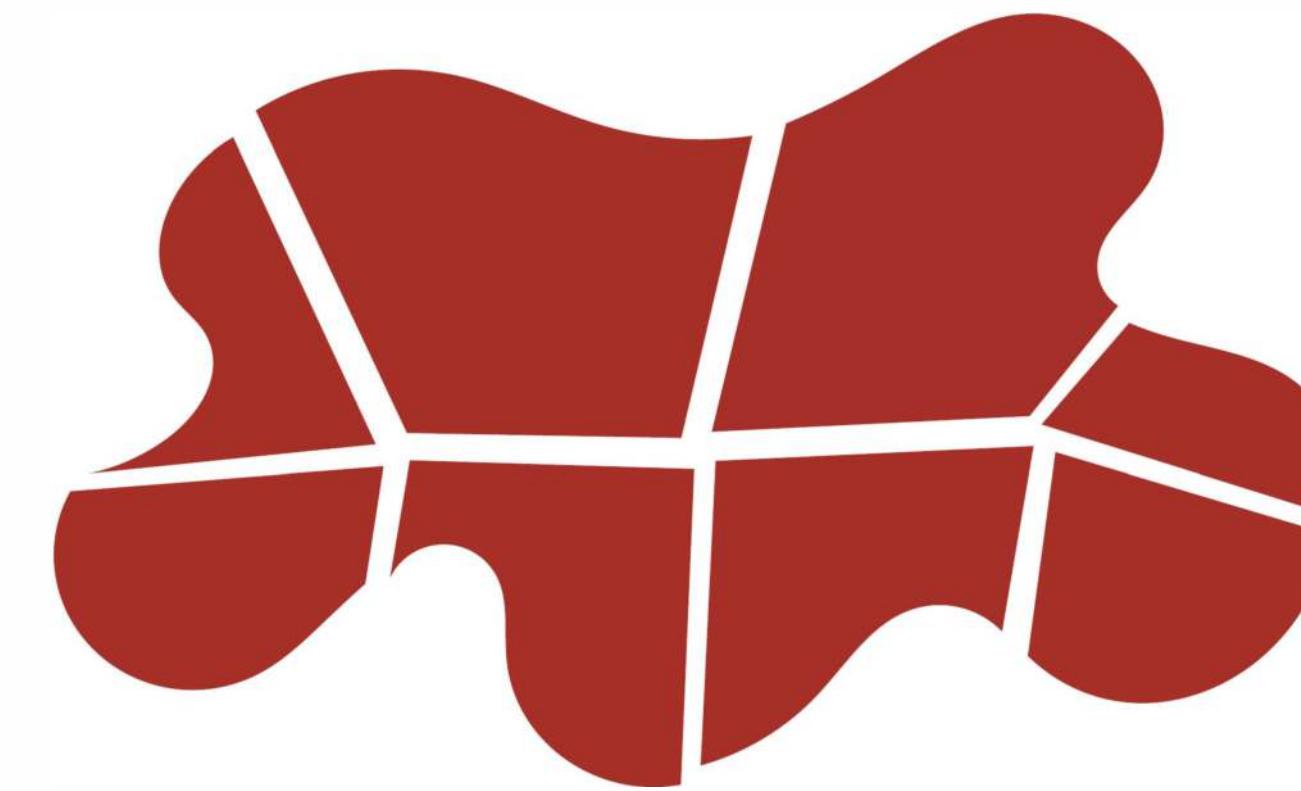


Chunk up the data...

Distributed Data-Parallel: High Level Illustration



Distribute it over your cluster of machines.



Distributed Data-Parallel: High Level Illustration



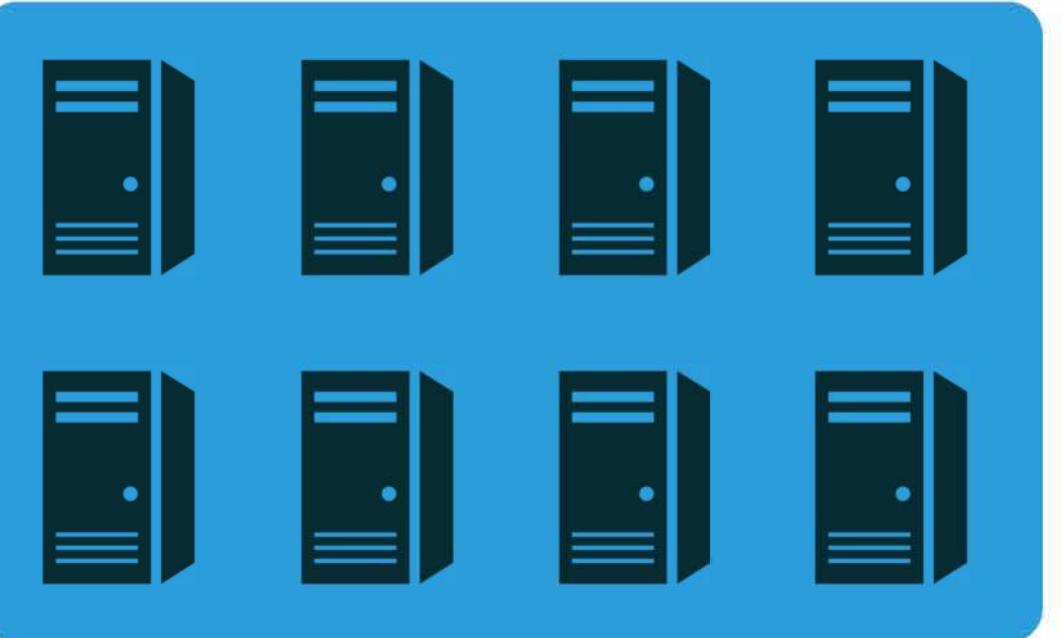
Distribute it over your cluster of machines.

Distributed Data-Parallel: High Level Illustration

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

Example:

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

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

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 <u>sequentially from memory</u>	<u>250,000ns</u> = 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	<u>20,000,000ns</u> = 20ms
Send packet US → Europe → US	150,000,000ns = 150ms

100x

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

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 <i>memory</i>	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: fastest
disk: slow
network: slowest

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	Disk	Network
L1 cache reference		
0.5 s		
—		
Main memory reference	Disk seek	Round trip within same datacenter
100 s	16.5 weeks	5.8 days
—		—
Read 1MB sequentially from memory	Read 1MB sequentially from disk	Send packet US→Eur→US
2.9 days	7.8 months	4.8 years
seconds/days	weeks/months	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: **100x 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!

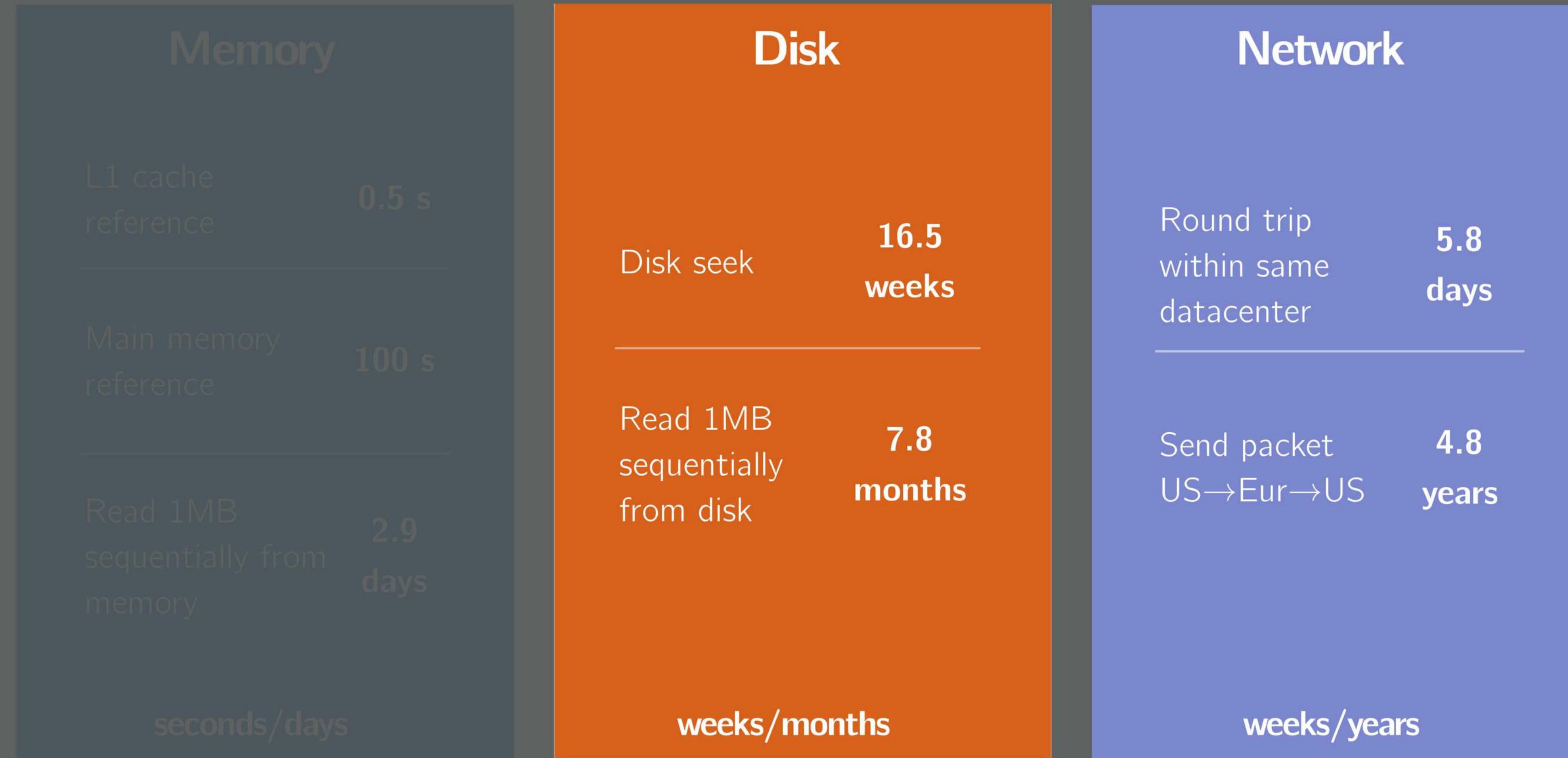
Idea: Keep all data **immutable** and **in-memory**. All operations on data are just functional transformations, like regular Scala collections. Fault tolerance is achieved by replaying functional transformations over original dataset.

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

Latency and System Design (Humanized)

Memory	Disk	Network
L1 cache reference	0.5 s	
Main memory reference	100 s	
Read 1MB sequentially from memory	2.9 days	
<hr/> <u>seconds/days</u> <hr/> <u>weeks/months</u> <hr/> <u>weeks/years</u>		
Disk seek	16.5 weeks	Round trip within same datacenter
Read 1MB sequentially from disk	7.8 months	Send packet US→Eur→US
		4.8 years

Latency and System Design



Hadoop

Latency and System Design

Memory	Disk	Network
L1 cache reference 0.5 s	Disk seek 16.5 weeks	Round trip within same datacenter 5.8 days
Main memory reference 100 s	Read 1MB sequentially from disk 7.8 months	Send packet US→Eur→US 4.8 years
Read 1MB sequentially from memory 2.9 days		

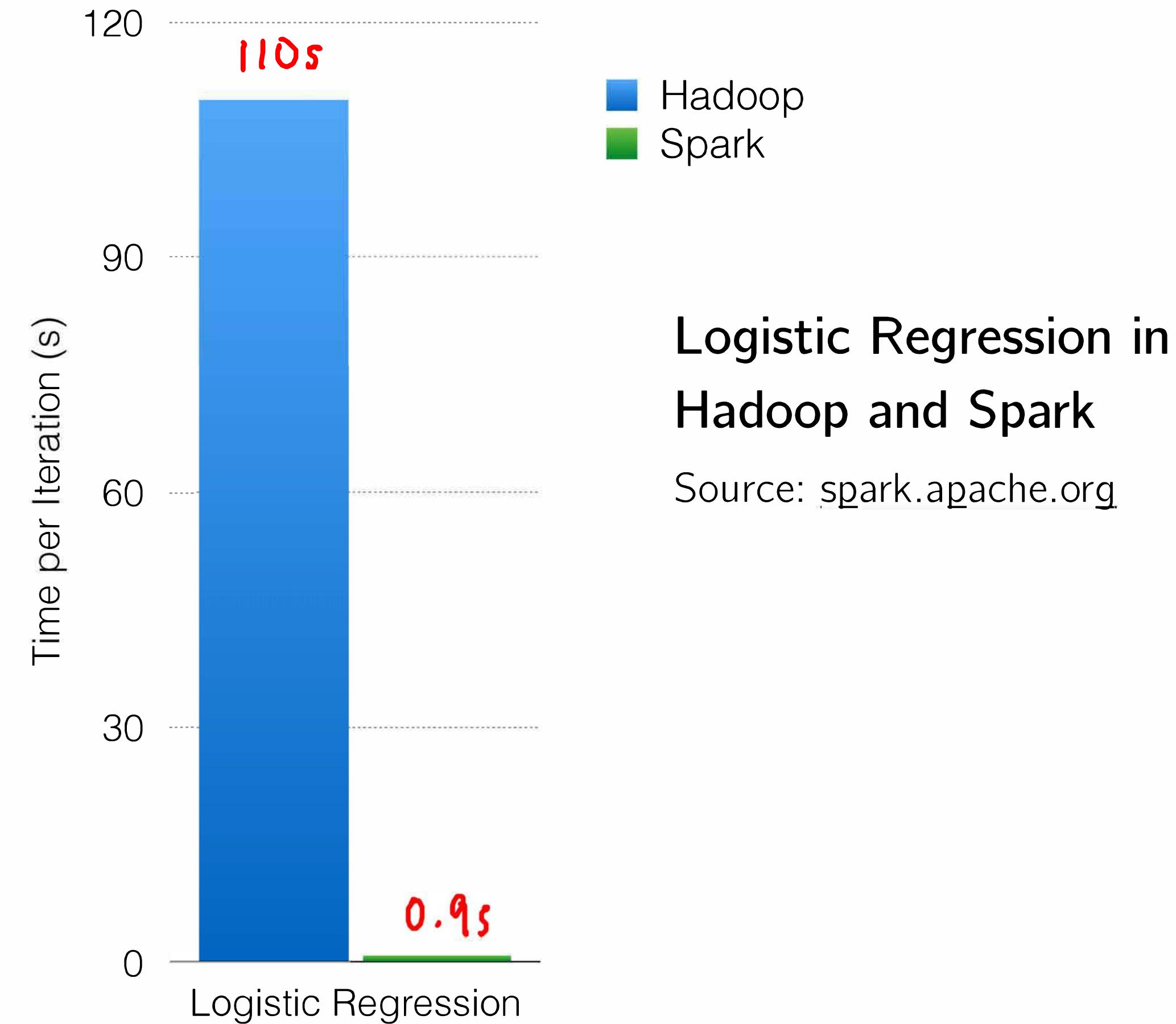
Spark



shift to inmemory

↑ aggressively minimize ↑

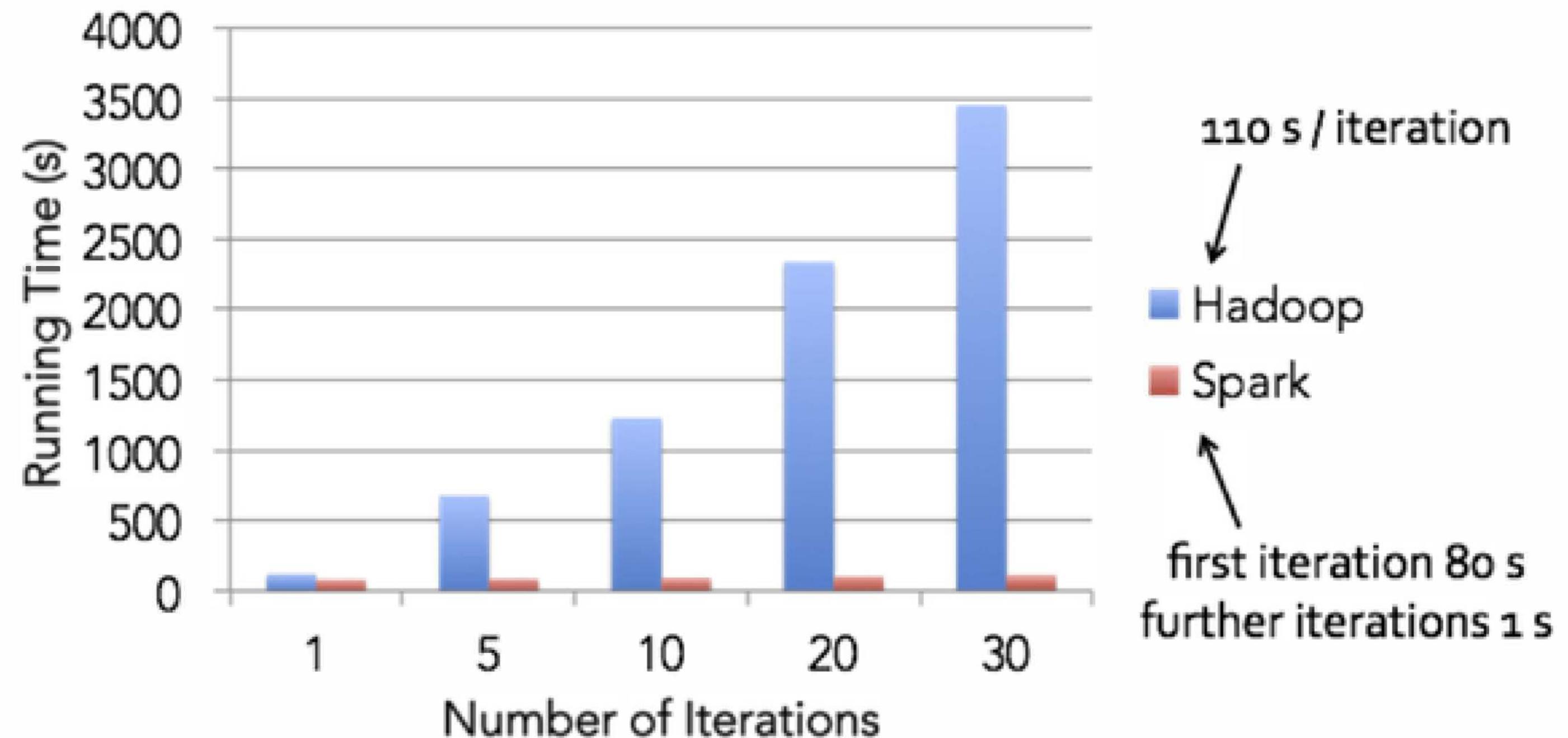
Spark versus Hadoop Performance?



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



Hadoop vs Spark Performance, More Intuitively

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

Hadoop/MapReduce

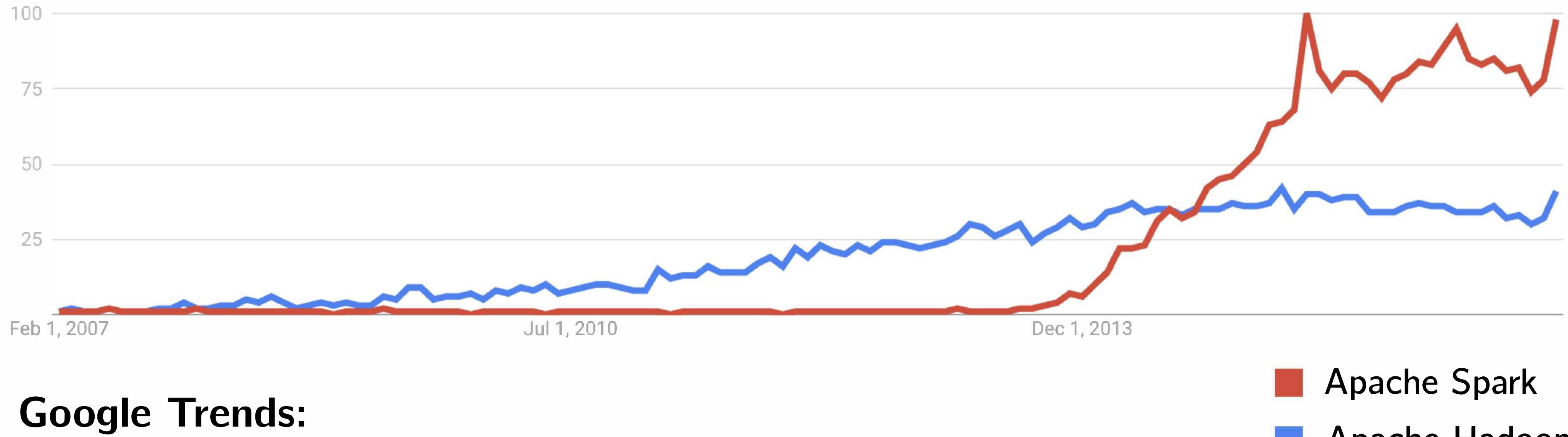
1. start job
 2. eat lunch
 3. get coffee
 4. pick up kids
 5. job completes
- (||)
= 'yo'

Spark

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

Spark versus Hadoop Popularity?

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



Google Trends:

Apache Hadoop vs Apache Spark

February 2007 - February 2017

- Apache Spark
- Apache Hadoop



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:

- map
- flatMap
- filter
- reduce
- fold
- aggregate

Combinators on RDDs:

- map
- flatMap
- filter
- reduce
- fold
- 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.

```
// Create an RDD      RDD[String]  
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://...")  
  
val count = rdd.flatMap(line => line.split(" ")) // separate lines into words
```

RDD[String] ← 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.

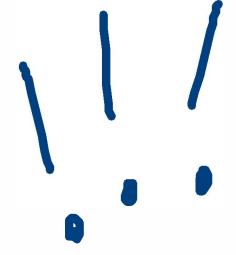
Transforming an existing RDD.

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

Big Data Analysis with Scala and Spark

Heather Miller

operations on
RDDs.

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.

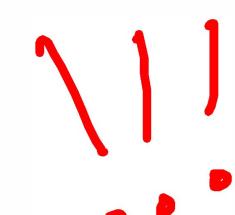
Transformations. Return new ~~collections~~ RDDs as results.

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.

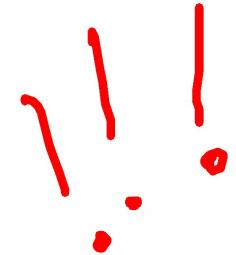
They seem similar to ~~transformers~~ and ~~accessors~~, but there are some important differences.

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

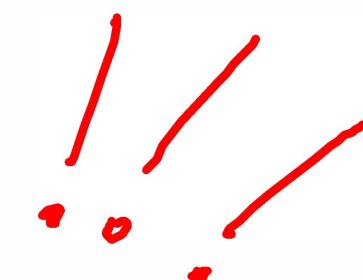
They are **lazy, their result RDD is not immediately computed.**

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)    RDD[String]  
val lengthsRdd = wordsRdd.map(_.length)      RDD[Int]
```

What has happened on the cluster at this point?

Example

Consider the following simple example:

```
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 return 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?

```
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.

Example:

filter
↓
take

```
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:

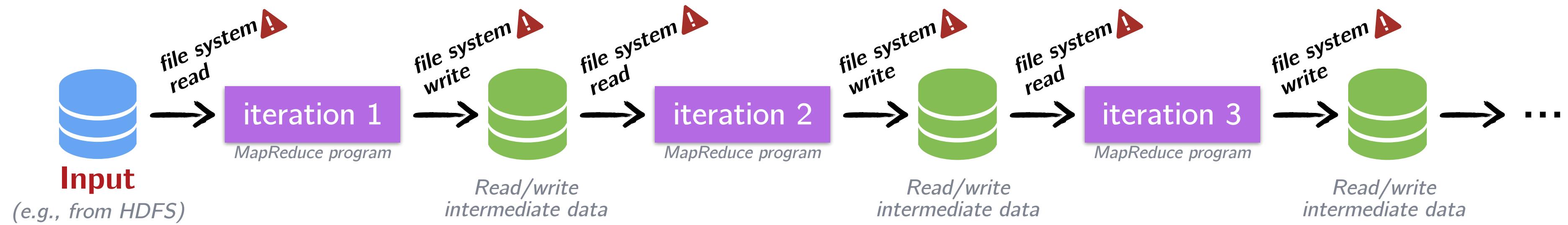
- ▶ We learned the difference between transformations and actions.
 - ▶ **Transformations:** Deferred/lazy
 - ▶ **Actions:** Eager, kick off staged transformations.
- ▶ We learned that latency makes a big difference; too much latency wastes the time of the data analyst.
 - ▶ **In-memory computation:**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

Why do you think Spark is good for data science?

Hint: Most data science problems involve iteration.

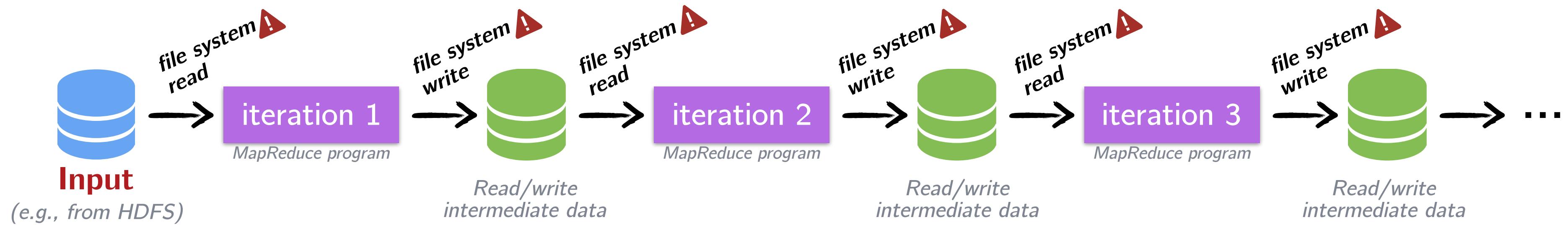
Iteration and Big Data Processing

Iteration in Hadoop:



Iteration and Big Data Processing

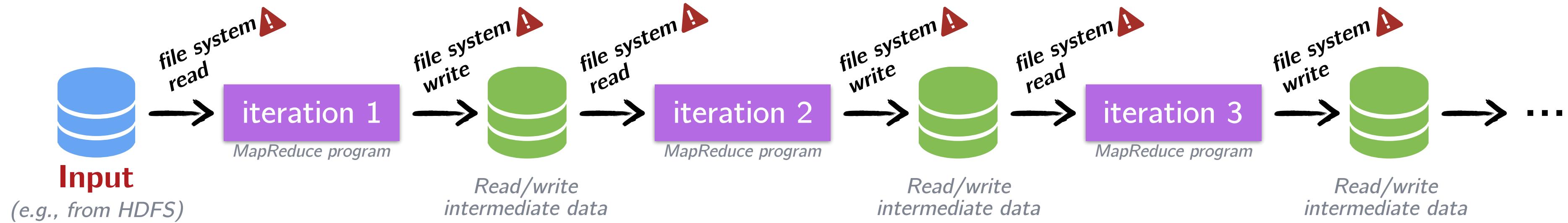
Iteration in Hadoop:



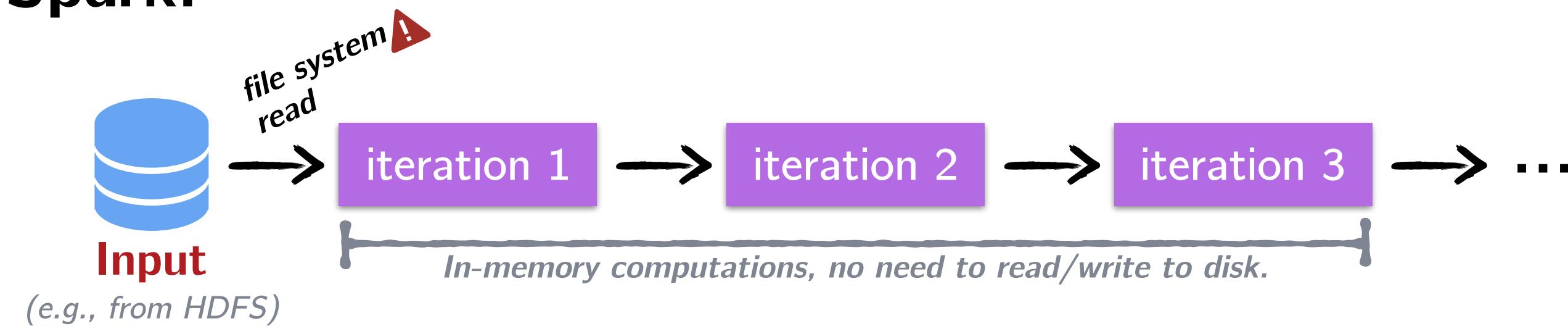
>90% of time in IO that Spark can avoid.

Iteration and Big Data Processing

Iteration in Hadoop:

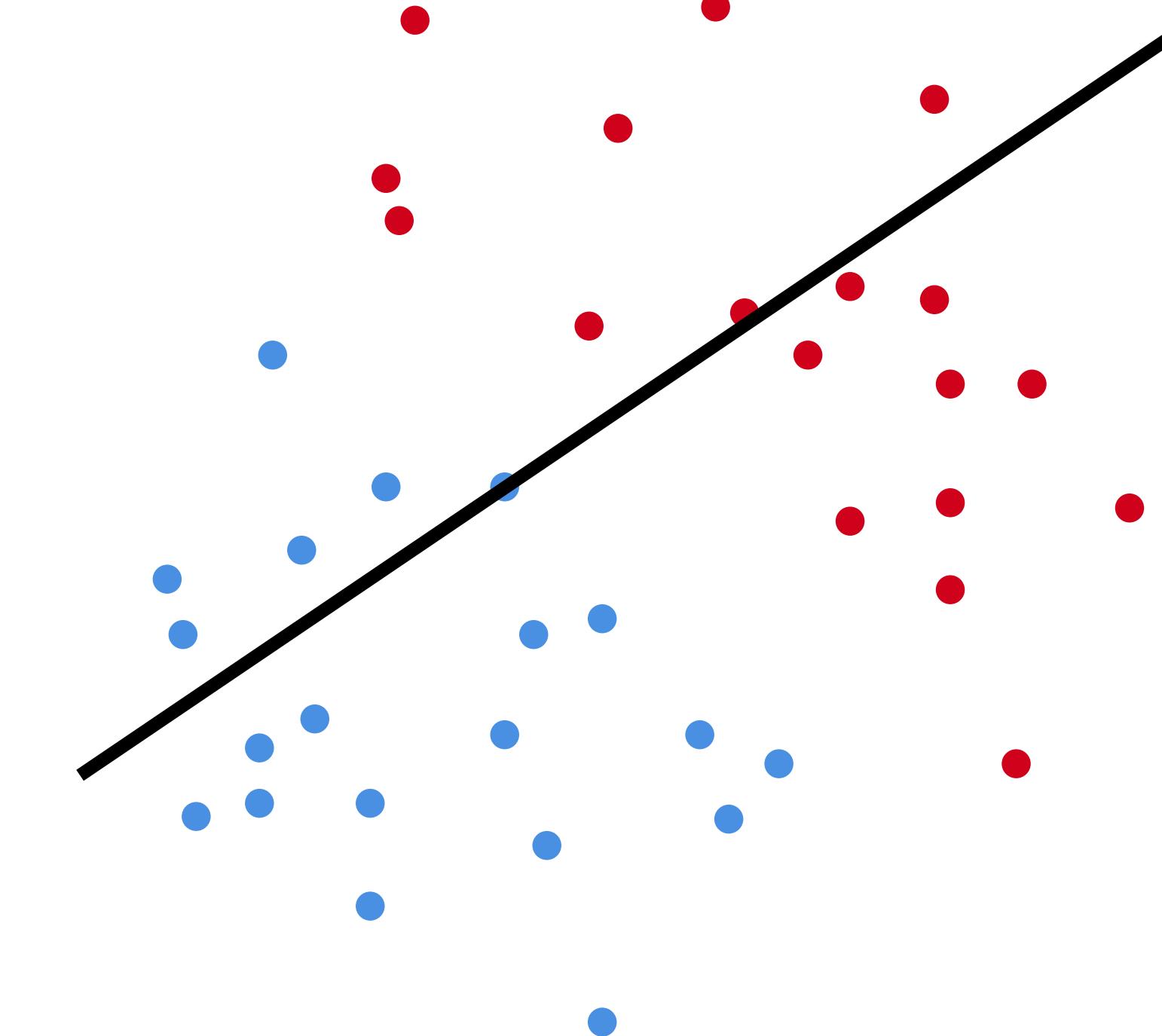


Iteration in Spark:



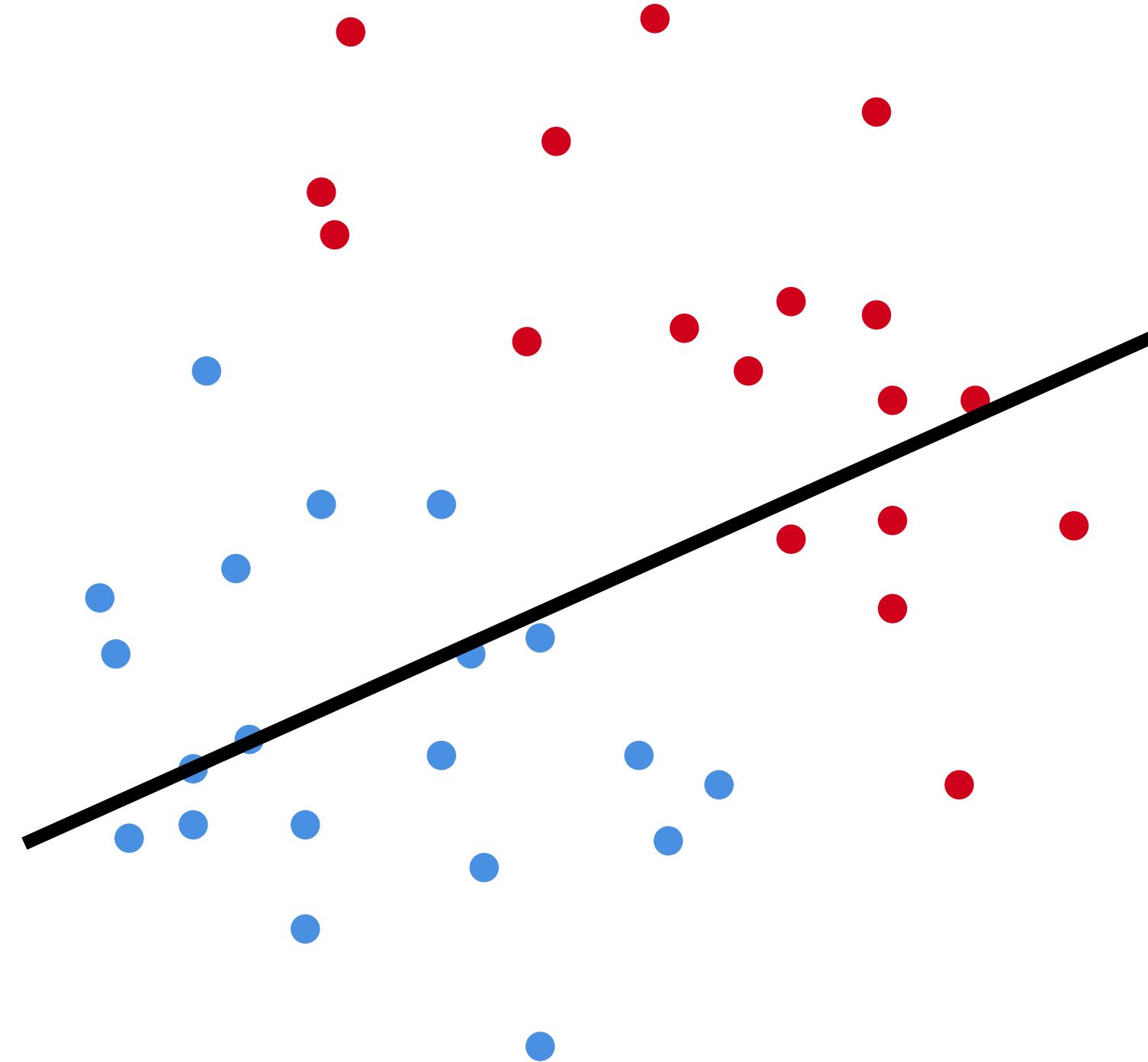
Iteration, Example: Logistic Regression

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.



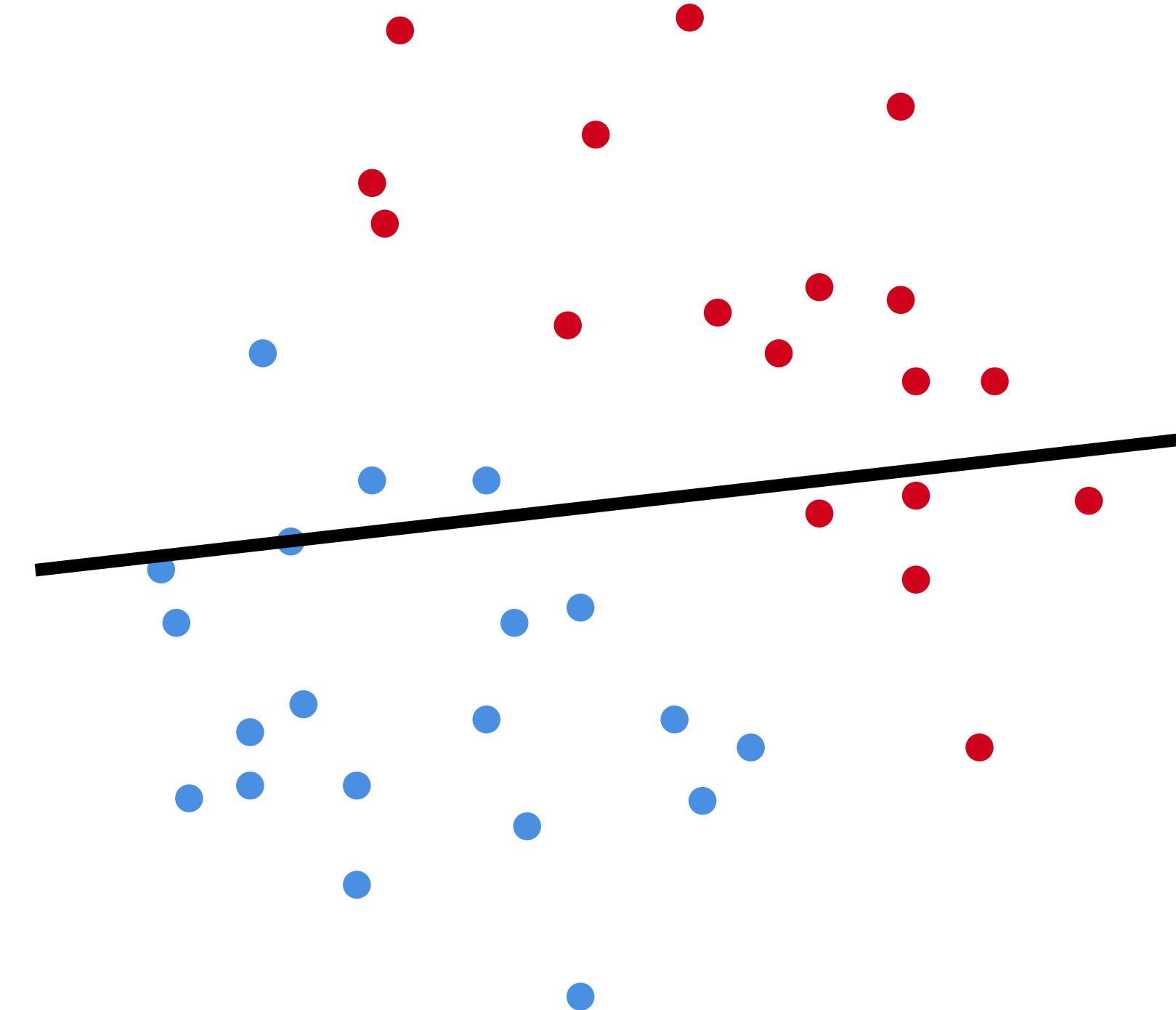
Iteration, Example: Logistic Regression

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.



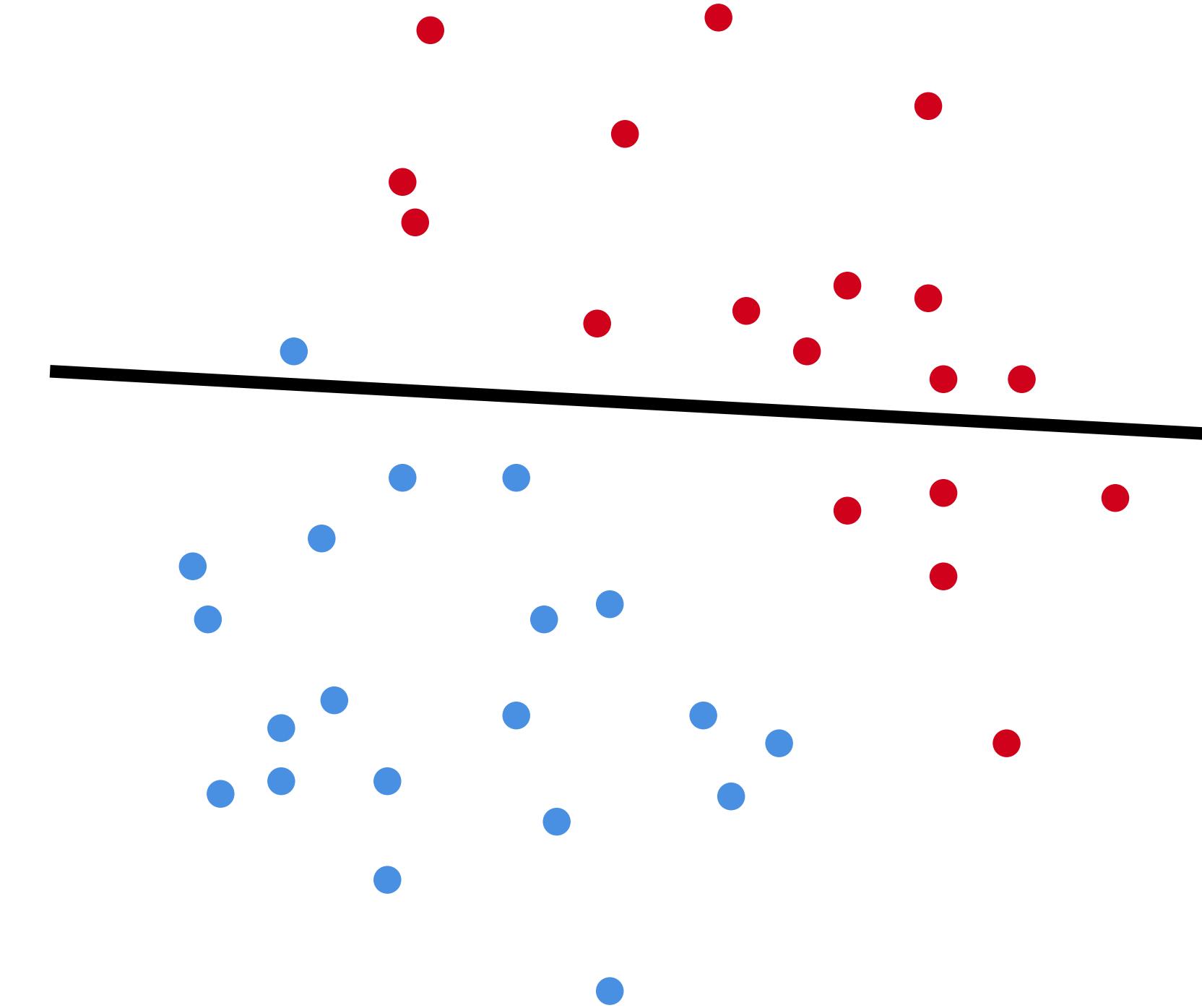
Iteration, Example: Logistic Regression

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.



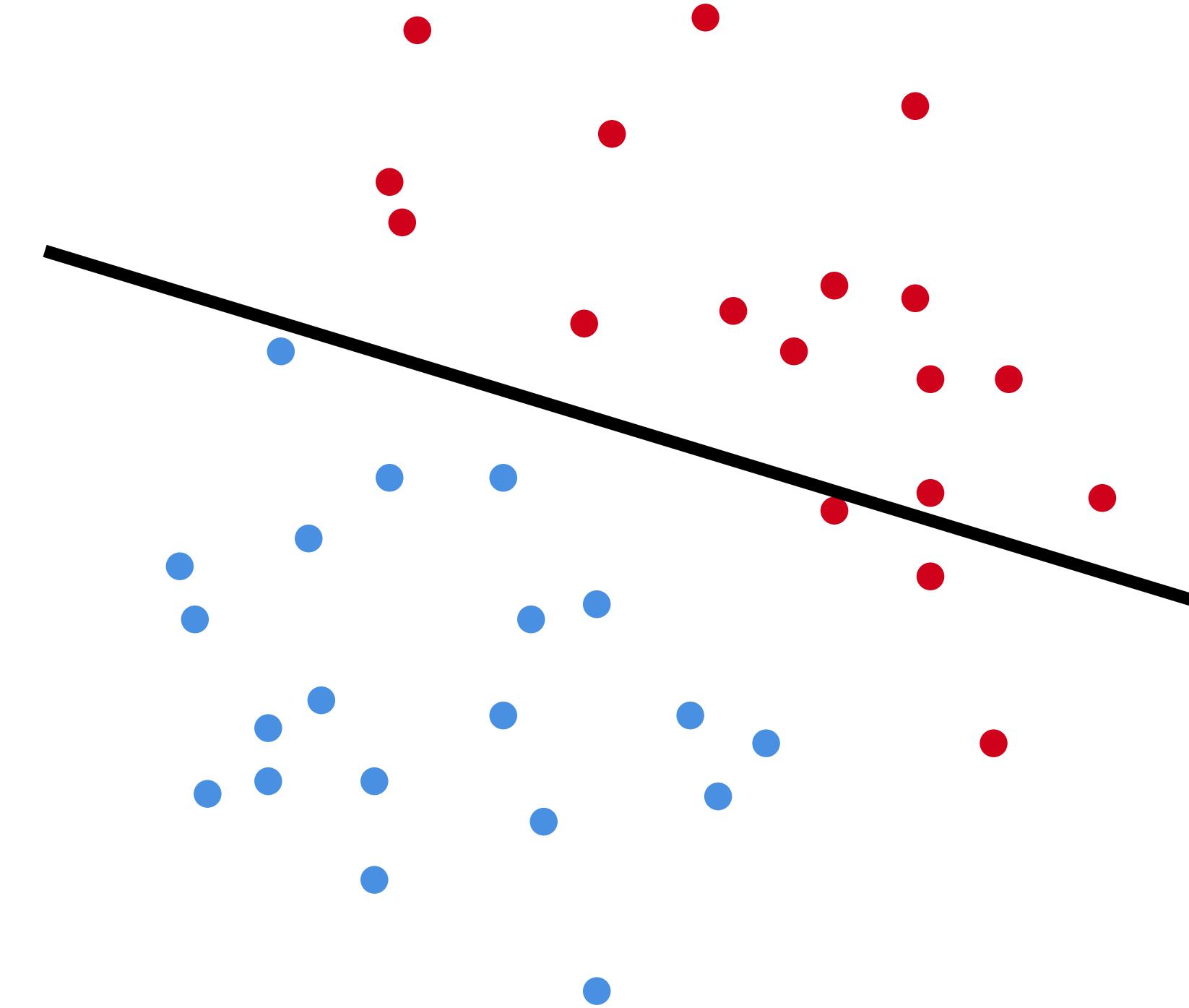
Iteration, Example: Logistic Regression

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.



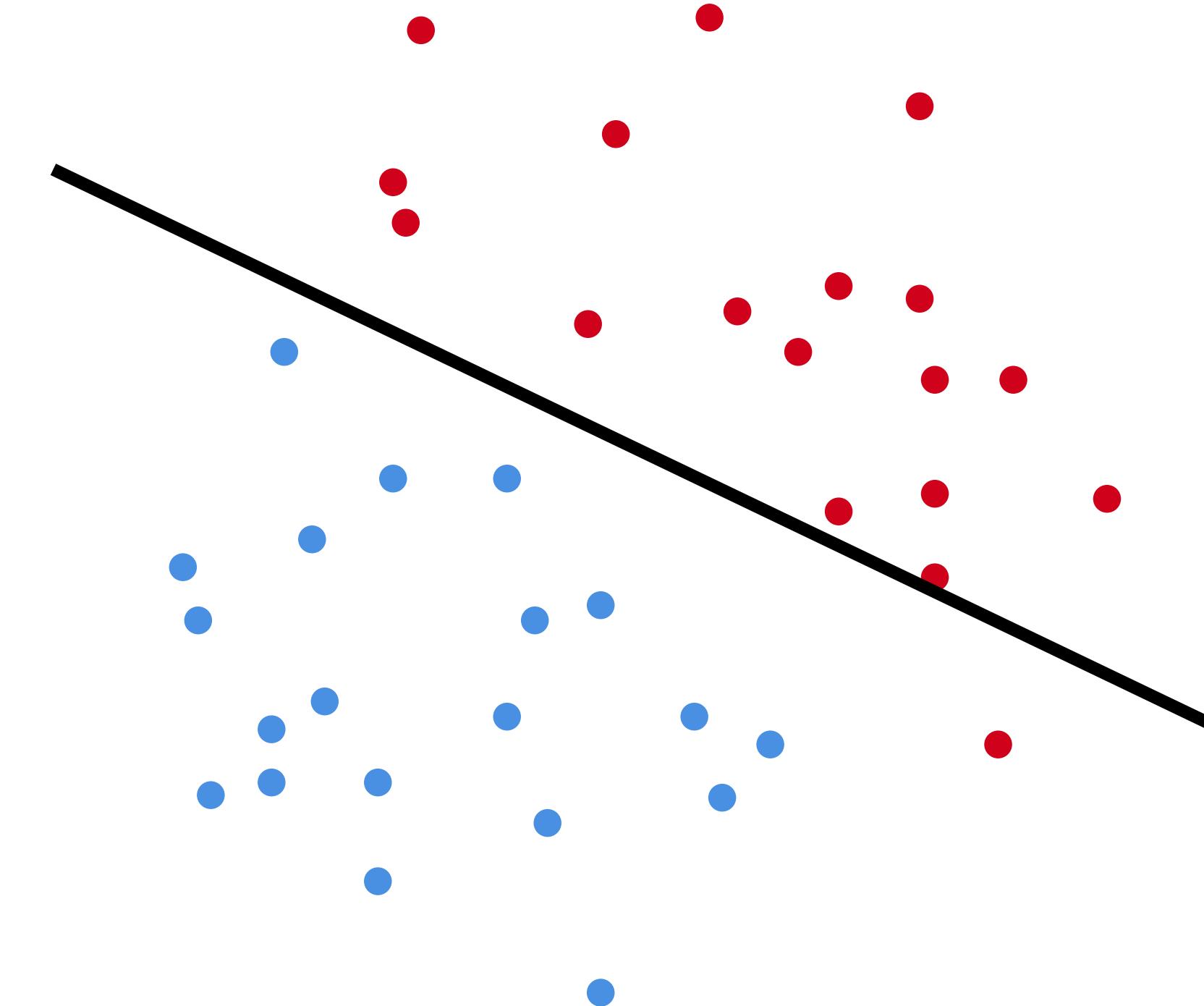
Iteration, Example: Logistic Regression

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.



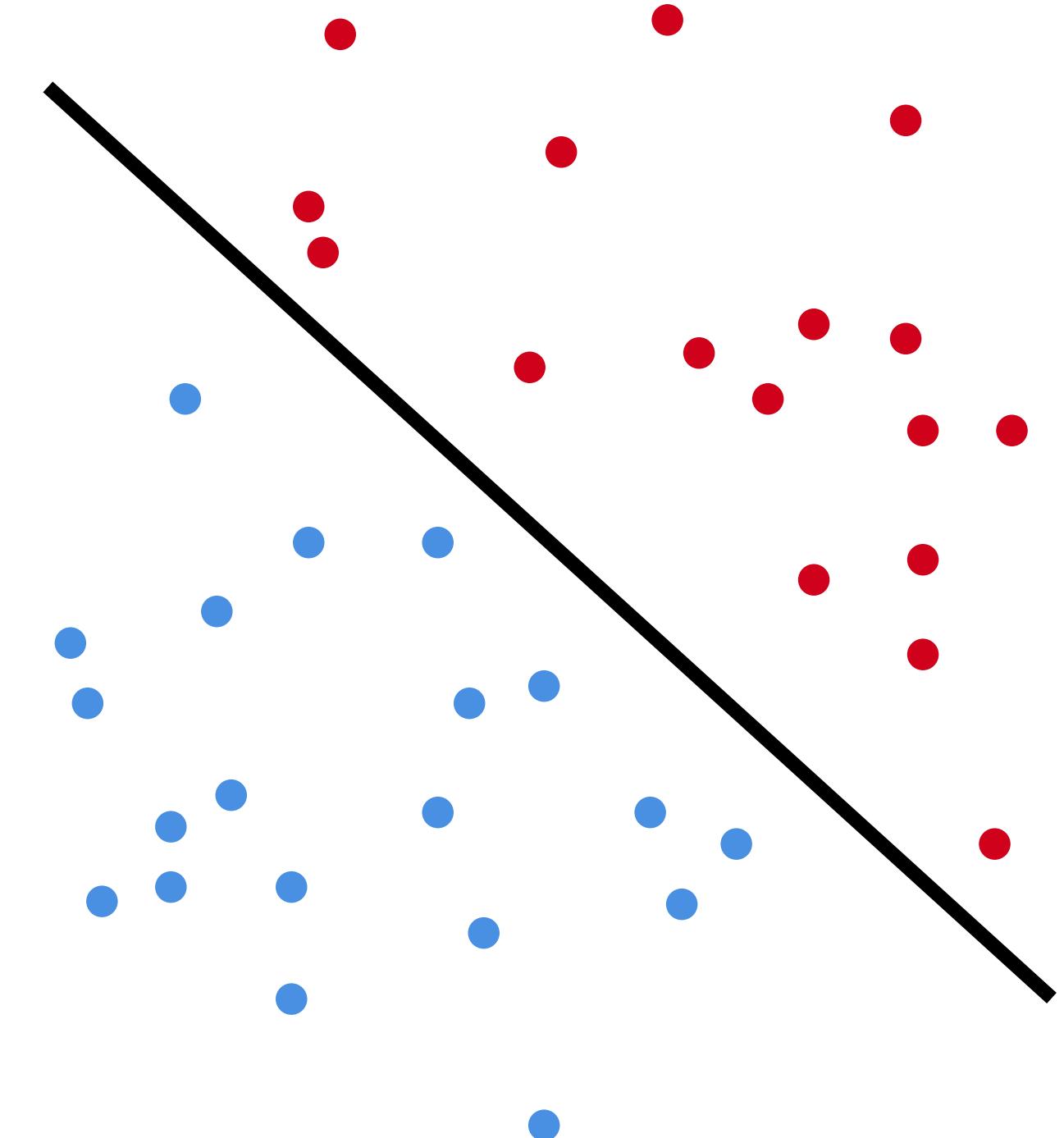
Iteration, Example: Logistic Regression

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:

```
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
}
```

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

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.

<i>Level</i>	<i>Space used</i>	<i>CPU time</i>	<i>In memory</i>	<i>On disk</i>
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.

<i>Level</i>	<i>Space used</i>	<i>CPU time</i>	<i>In memory</i>	<i>On disk</i>
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(_.toLowerCase)
                      .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:

```
val lastYearsLogs: RDD[String] = ...  
val numErrors = lastYearsLogs.map(_.toLowerCase)  
    .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:

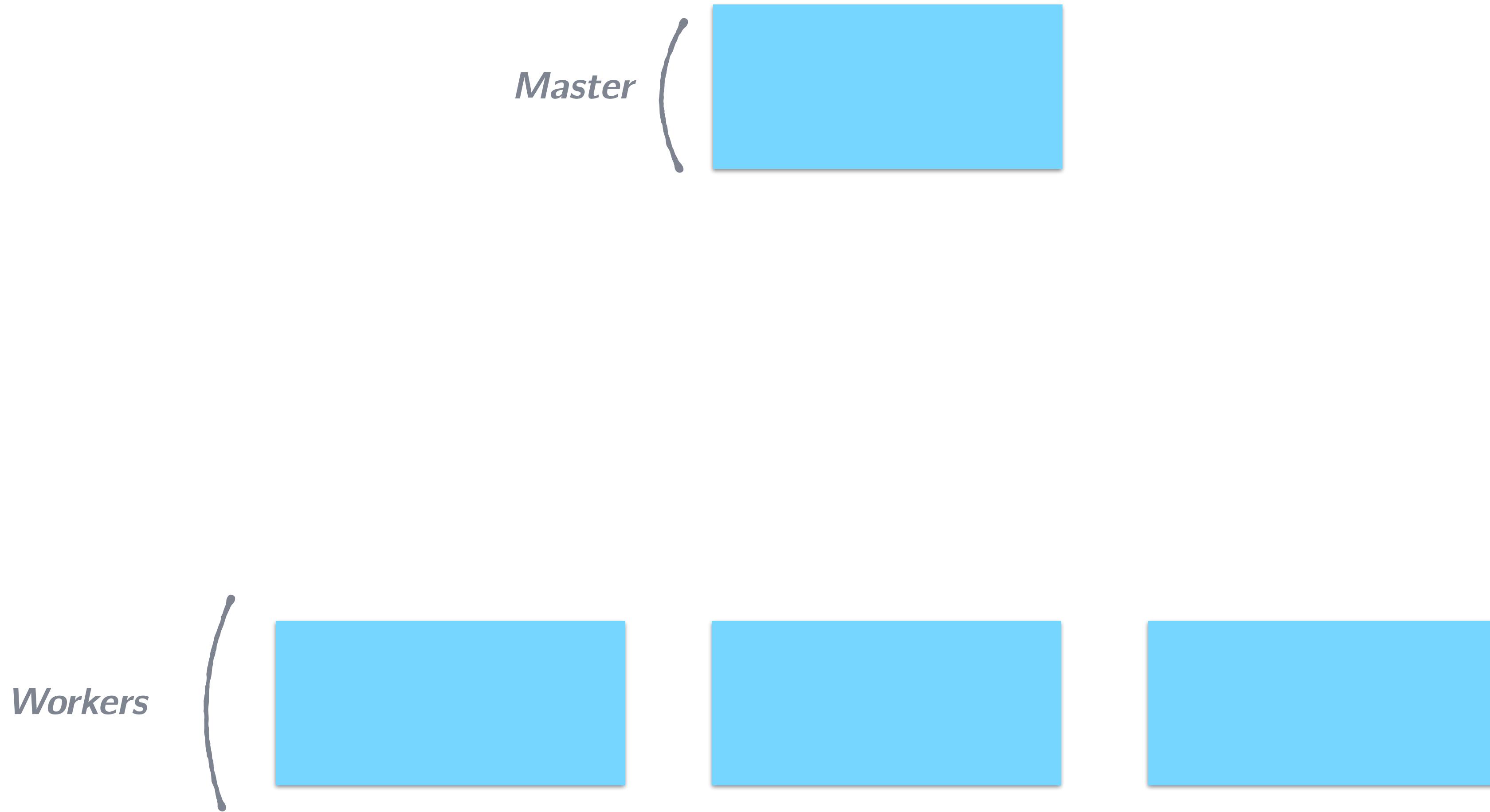
```
case class Person(name: String, age: Int)
```

What does the following code snippet do?

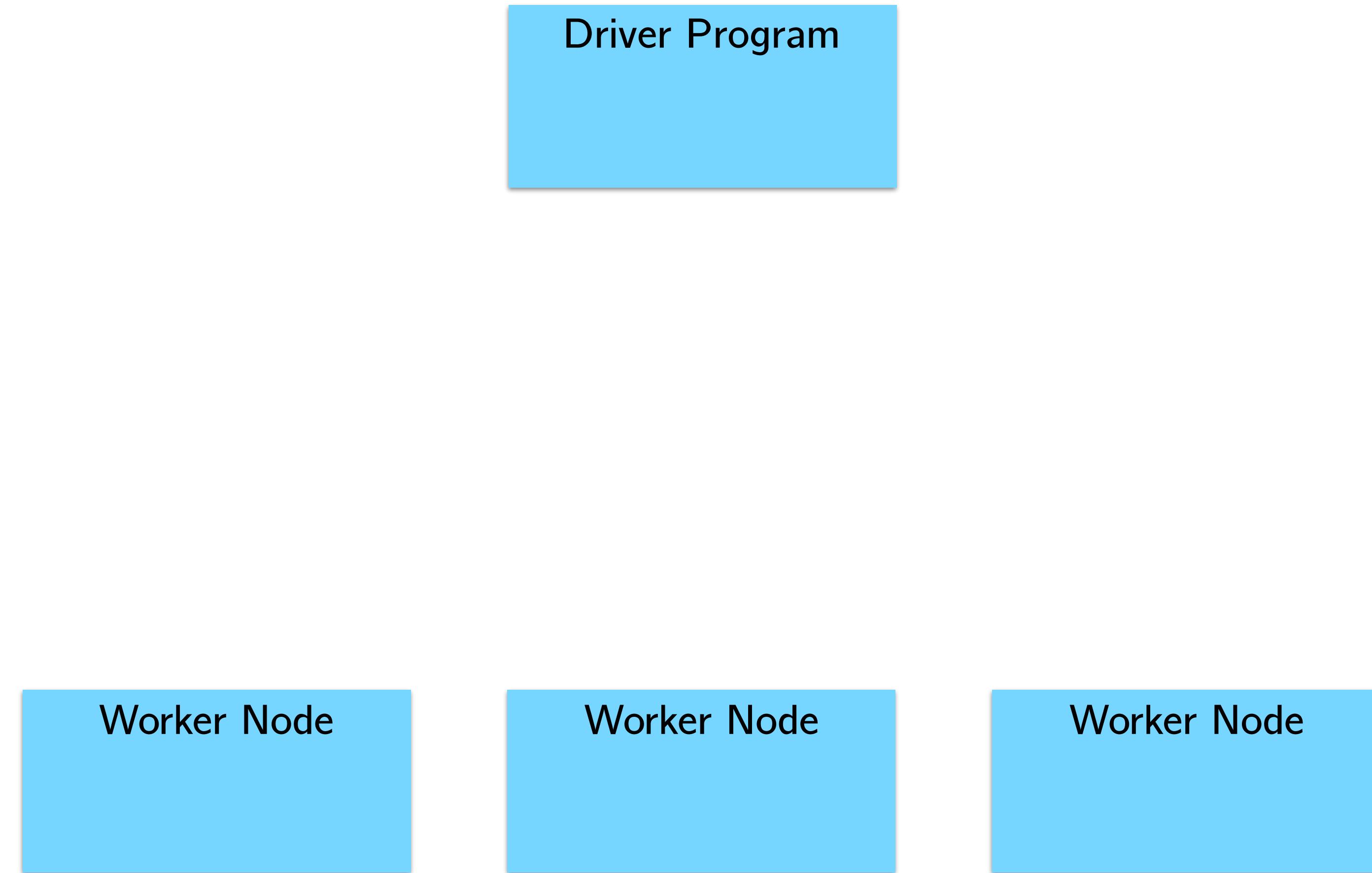
```
val people: RDD[Person] = ...
val first10 = people.take(10)
```

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

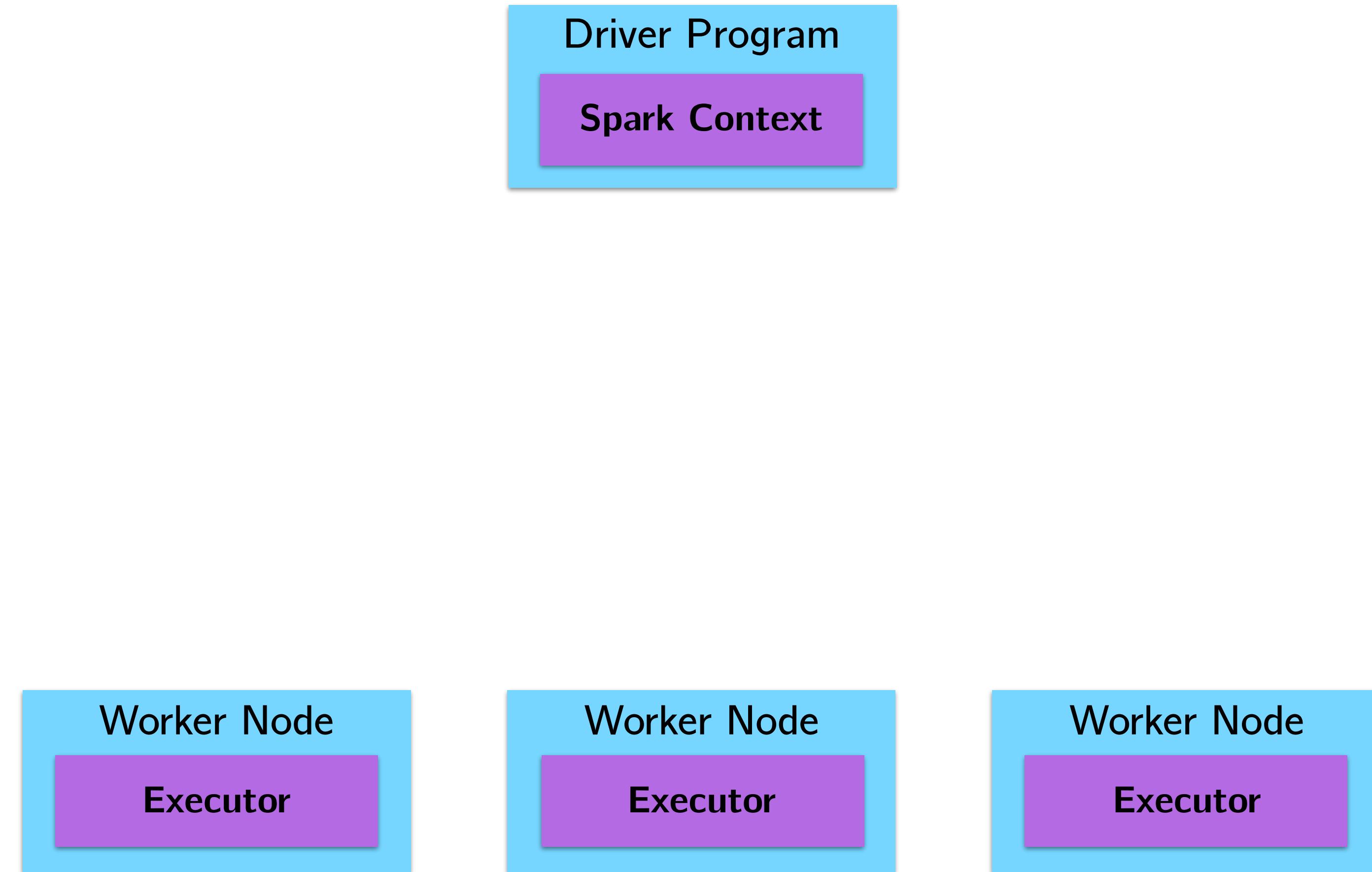
How Spark Jobs are Executed



How Spark Jobs are Executed

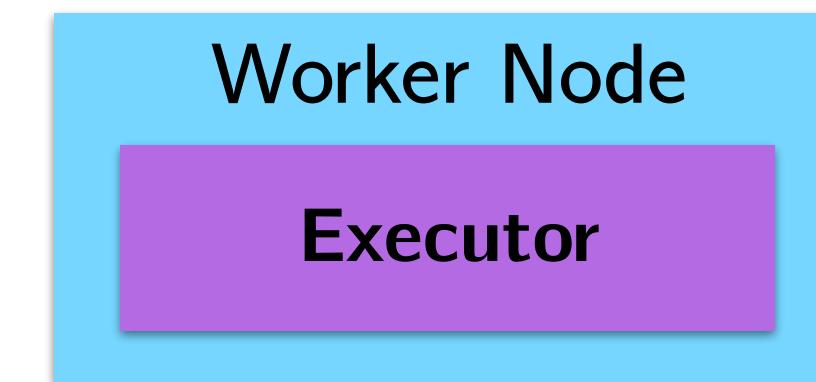
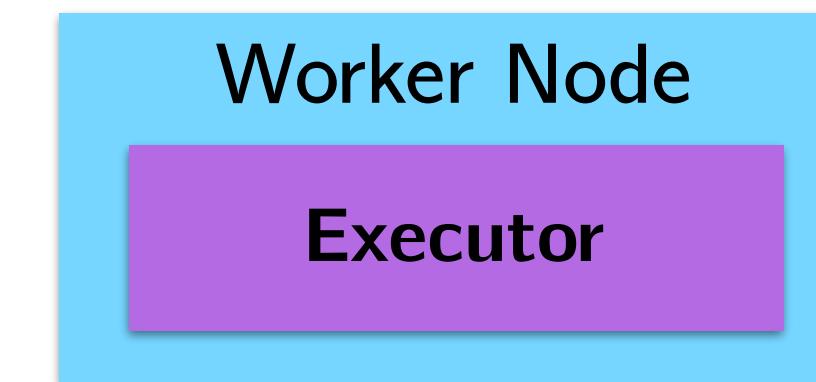
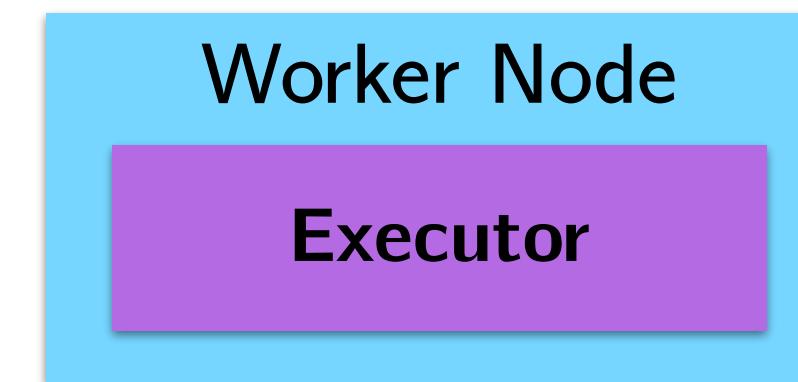
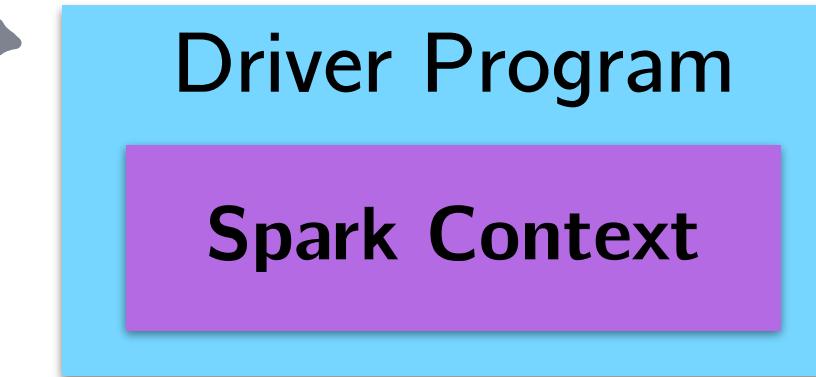


How Spark Jobs are Executed



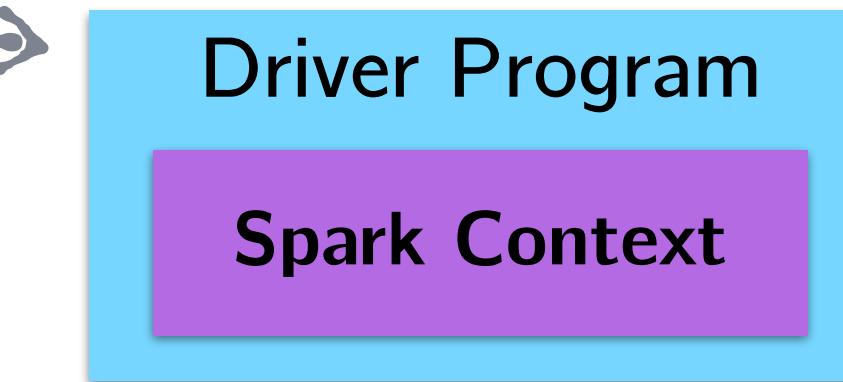
How Spark Jobs are Executed

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

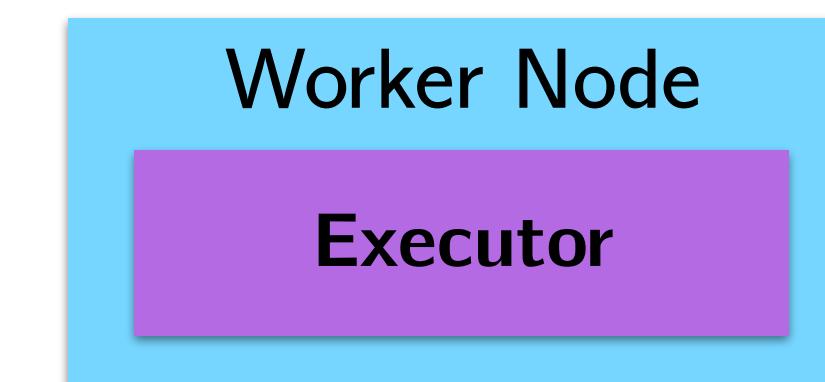
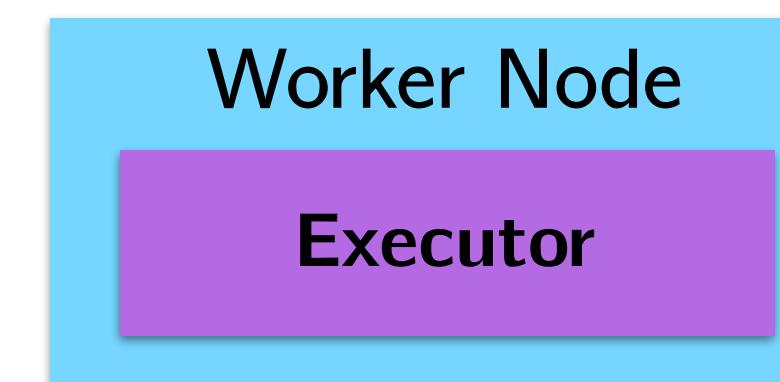
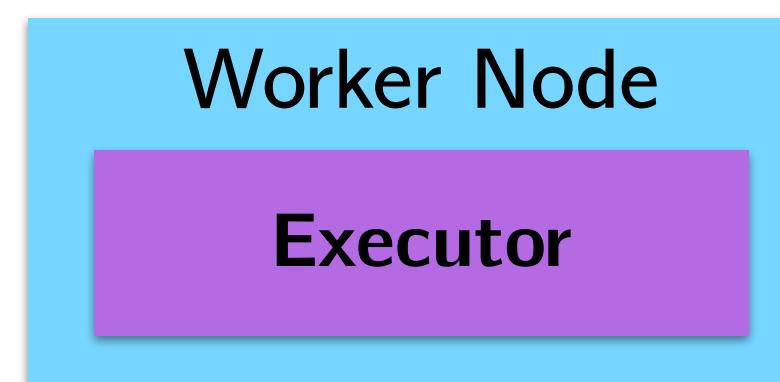


How Spark Jobs are Executed

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

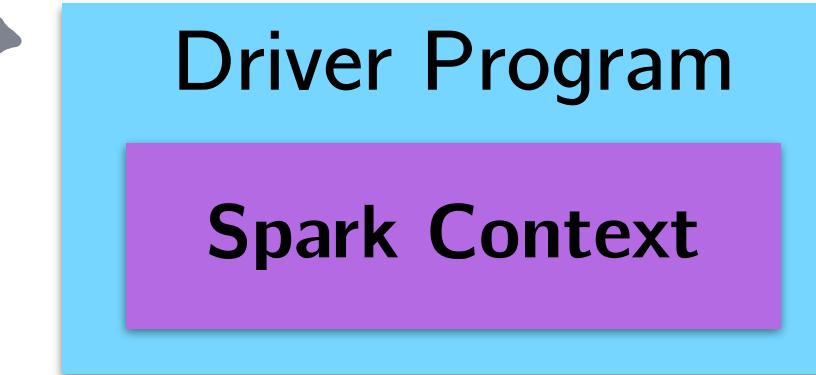


These are the nodes actually executing the jobs!



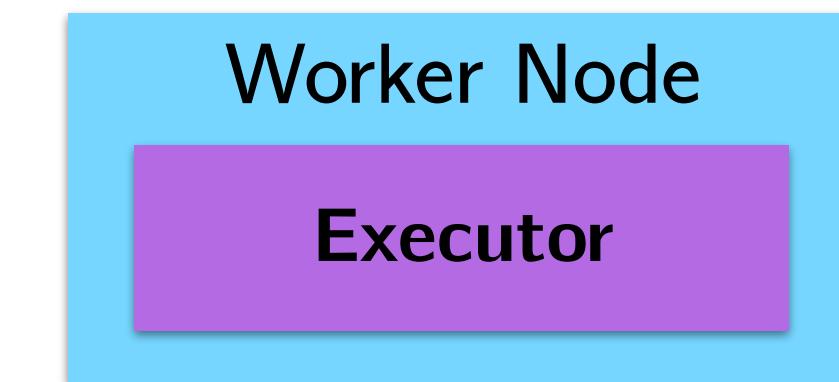
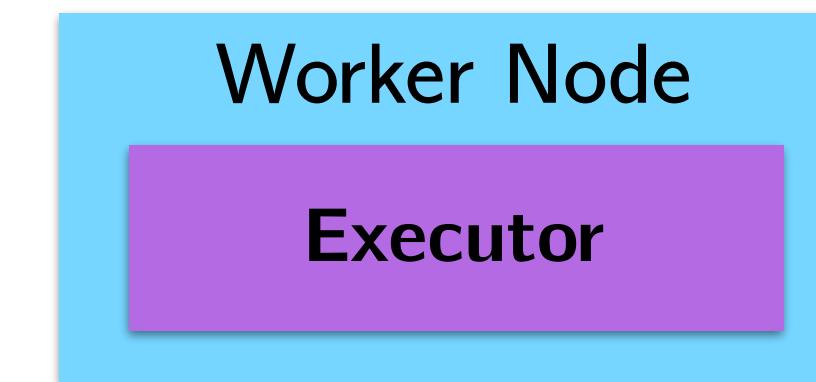
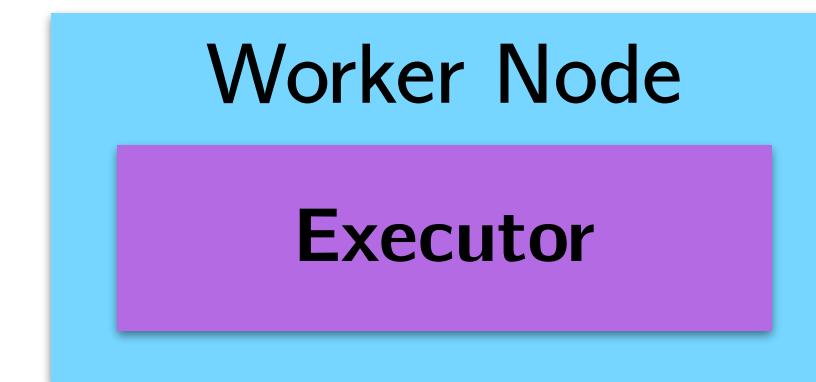
How Spark Jobs are Executed

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

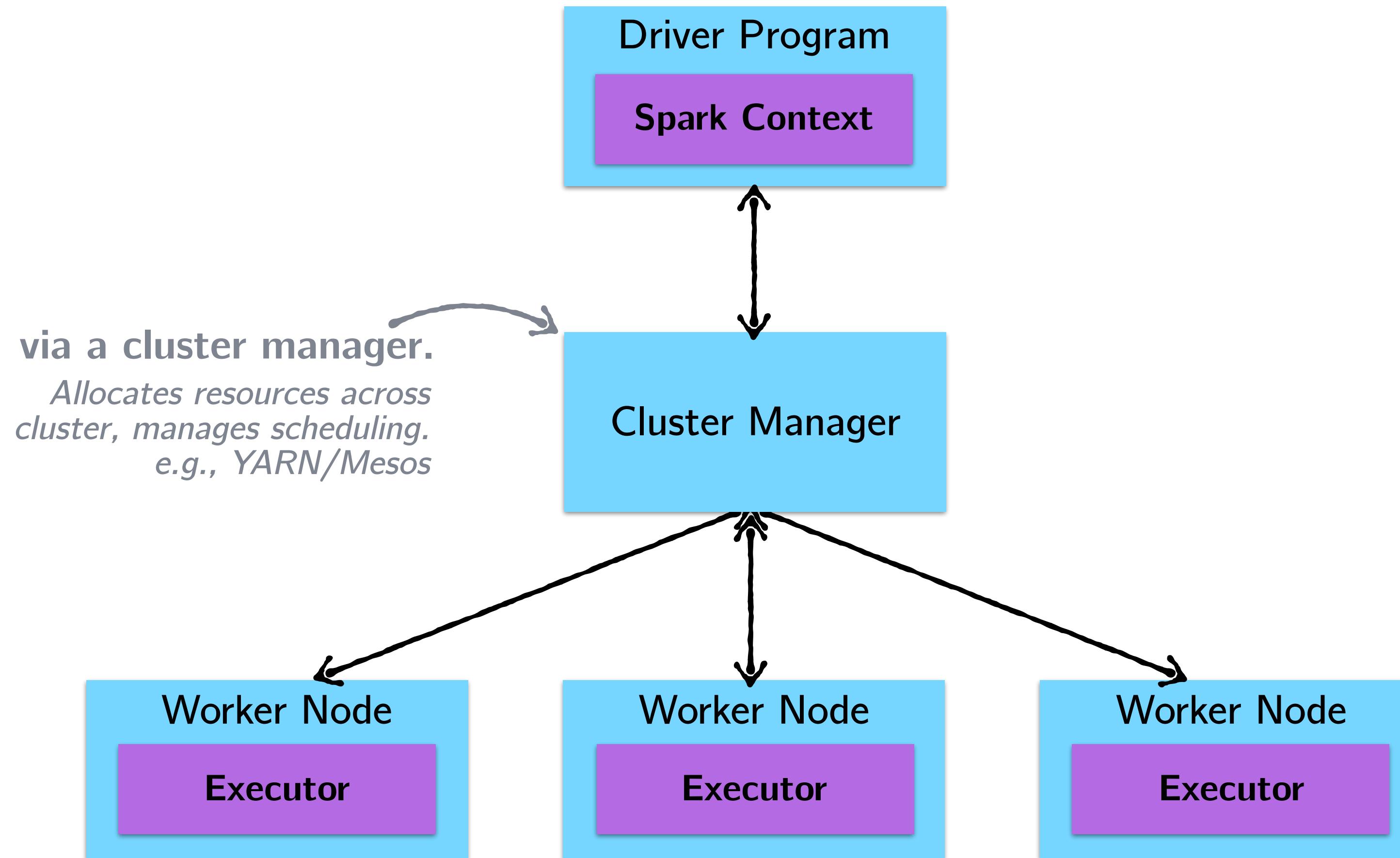


But how do they all communicate?

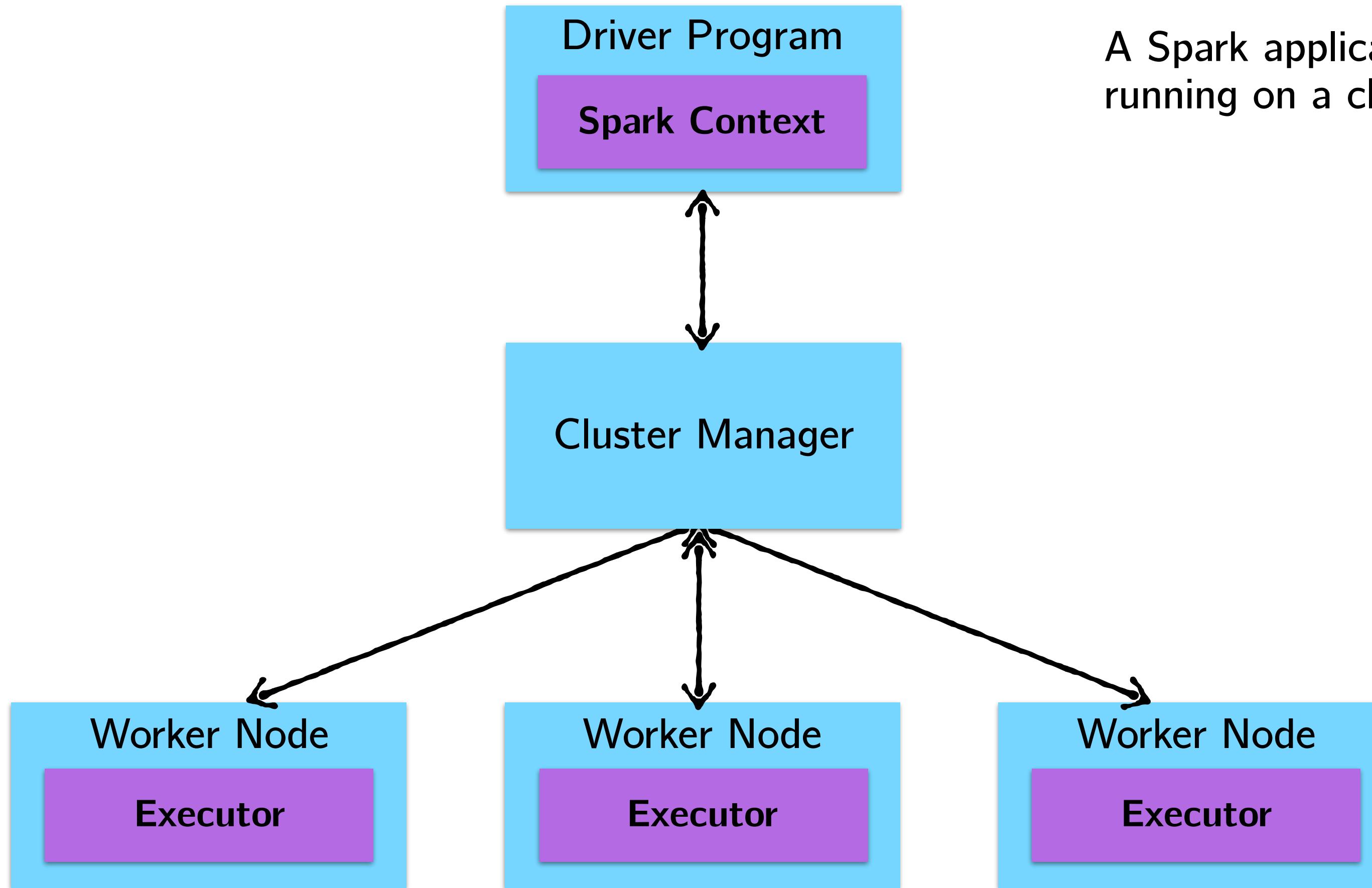
These are the nodes actually executing the jobs!



How Spark Jobs are Executed

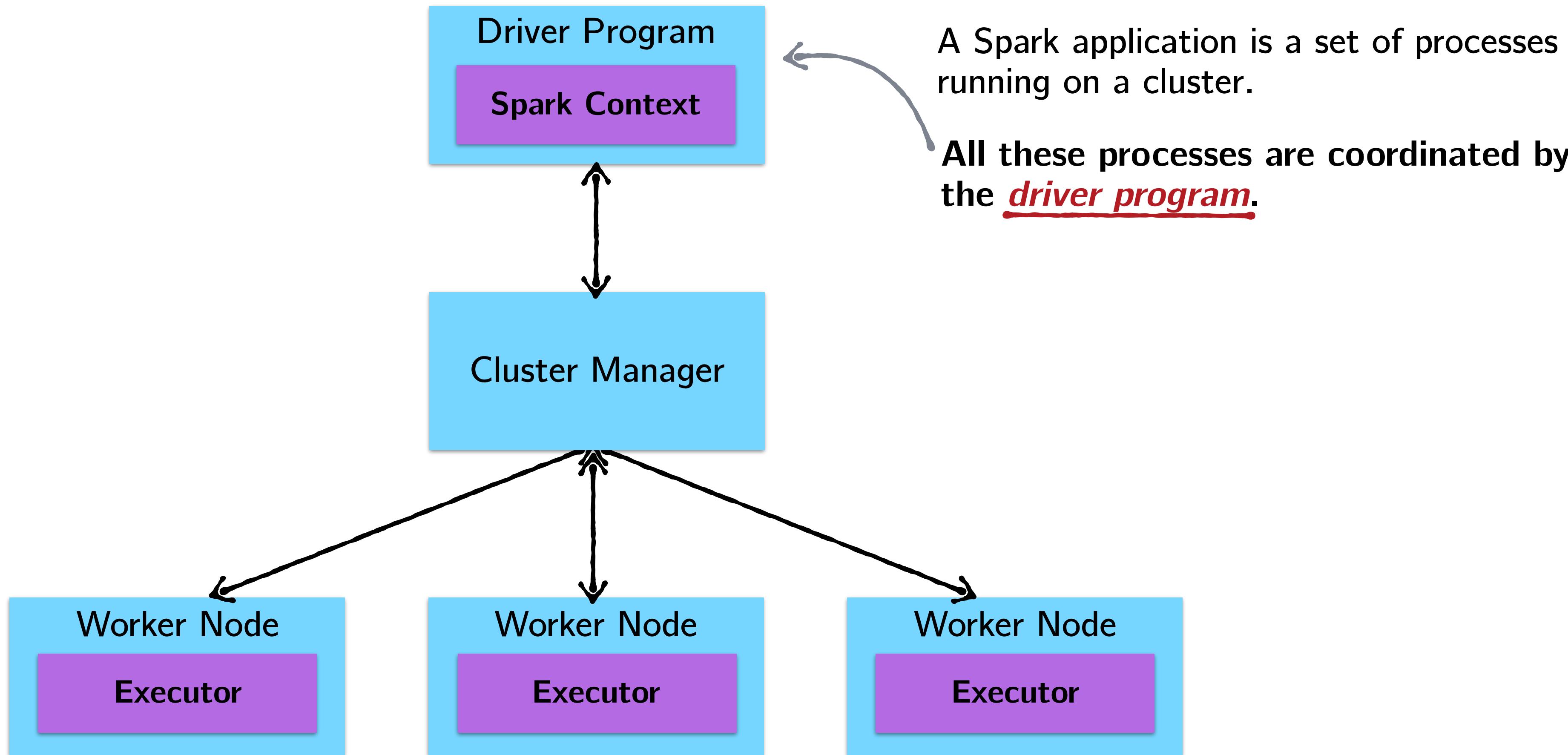


How Spark Jobs are Executed

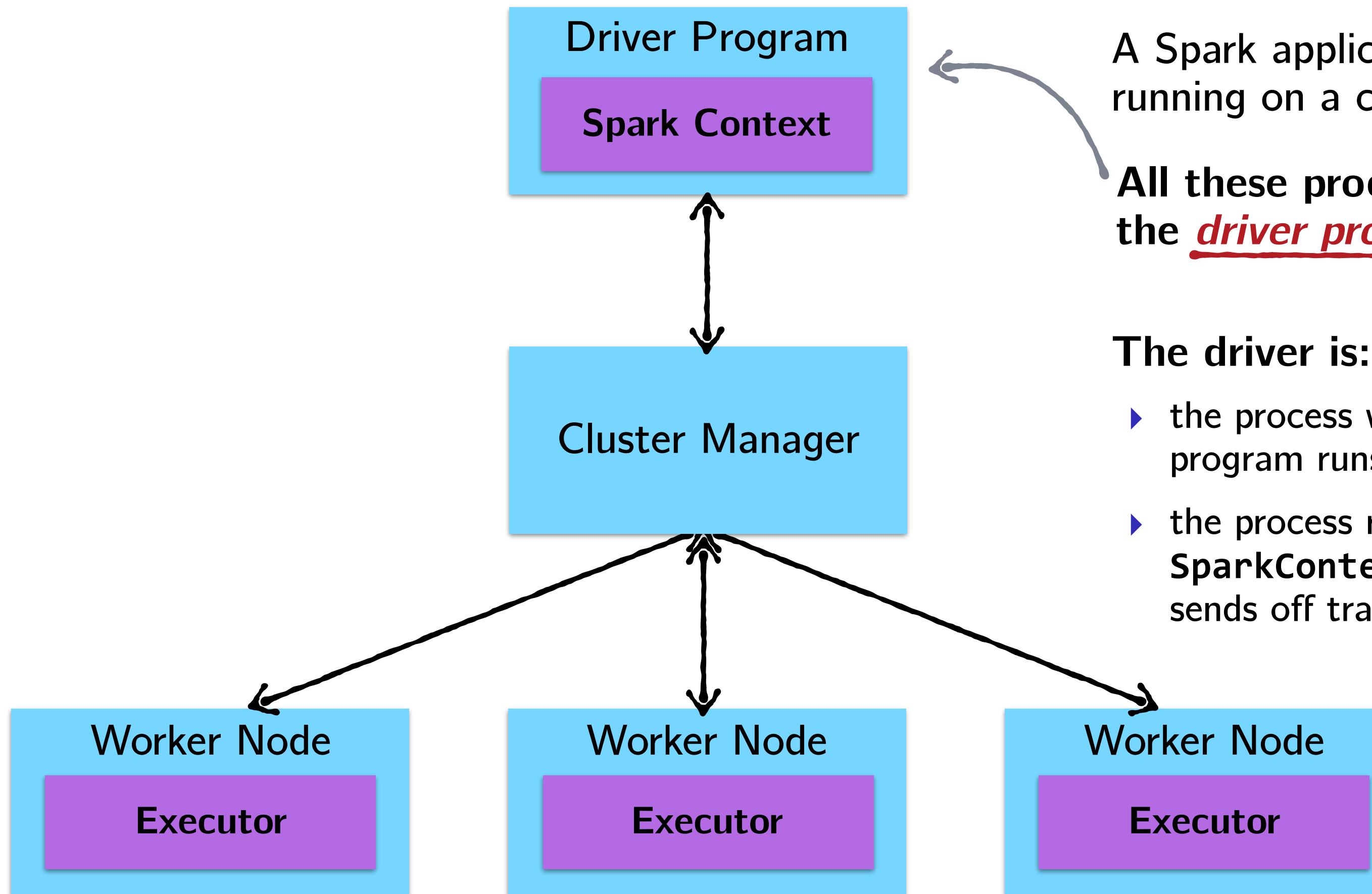


A Spark application is a set of processes running on a cluster.

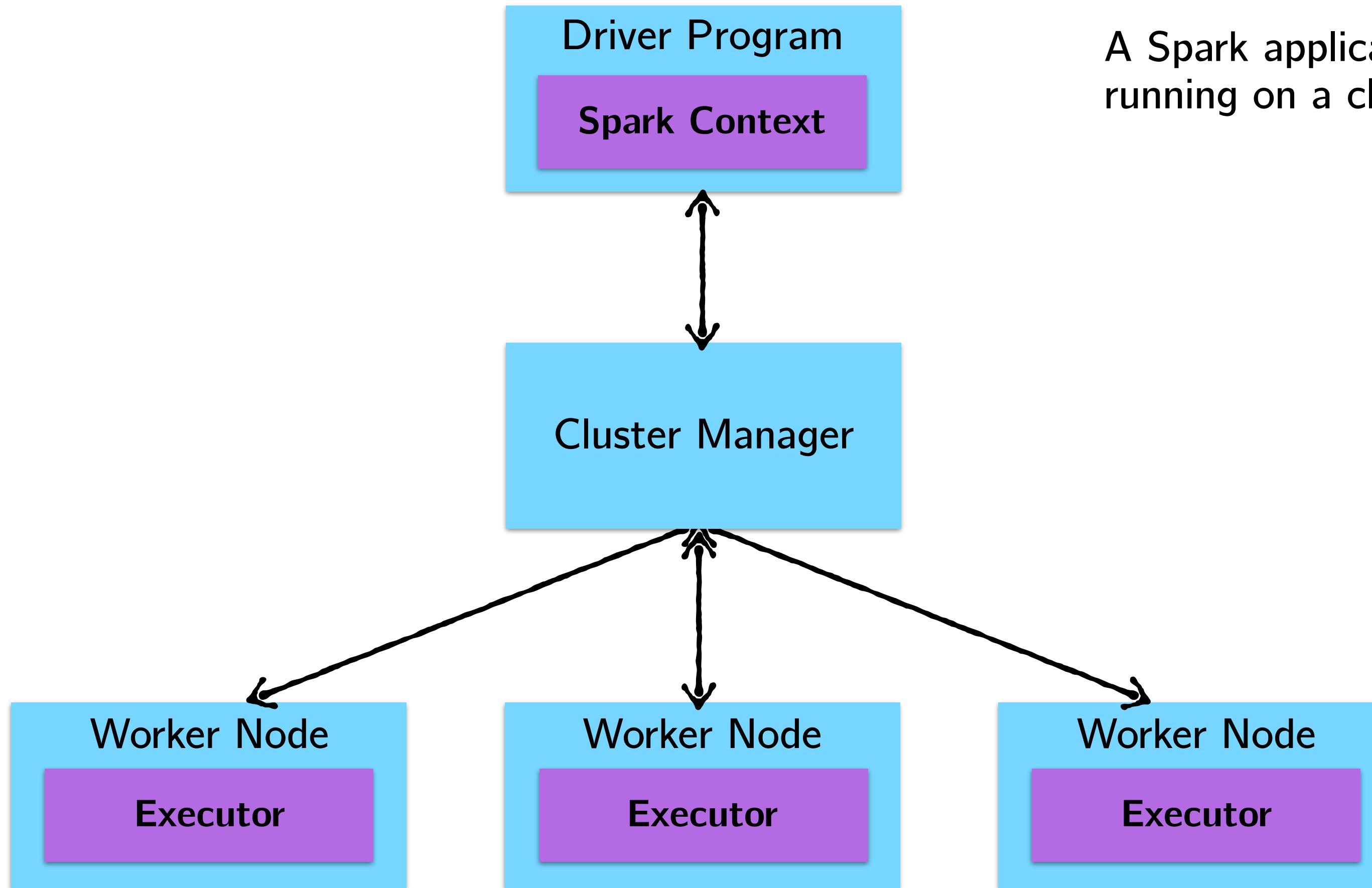
How Spark Jobs are Executed



How Spark Jobs are Executed

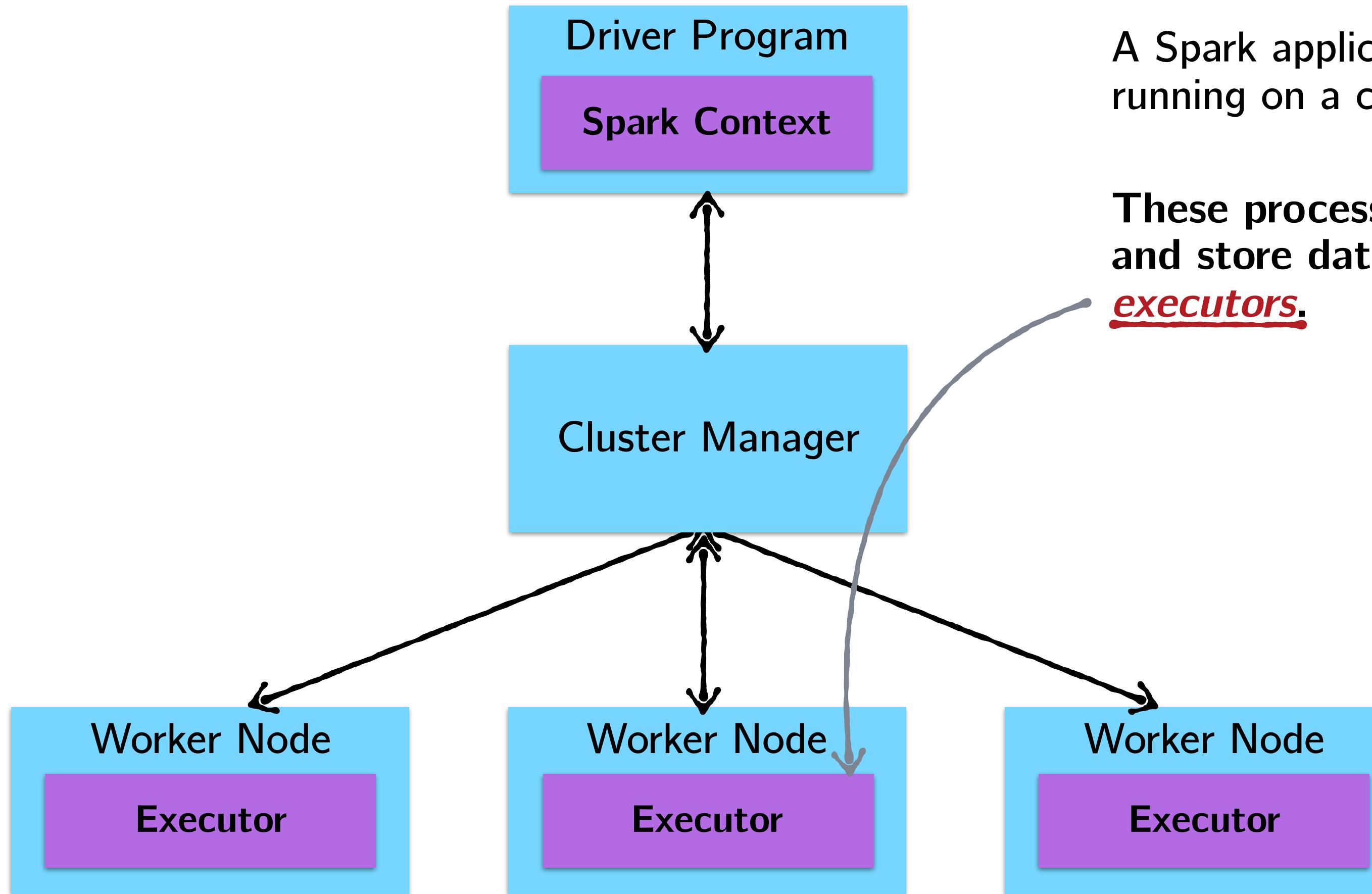


How Spark Jobs are Executed



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

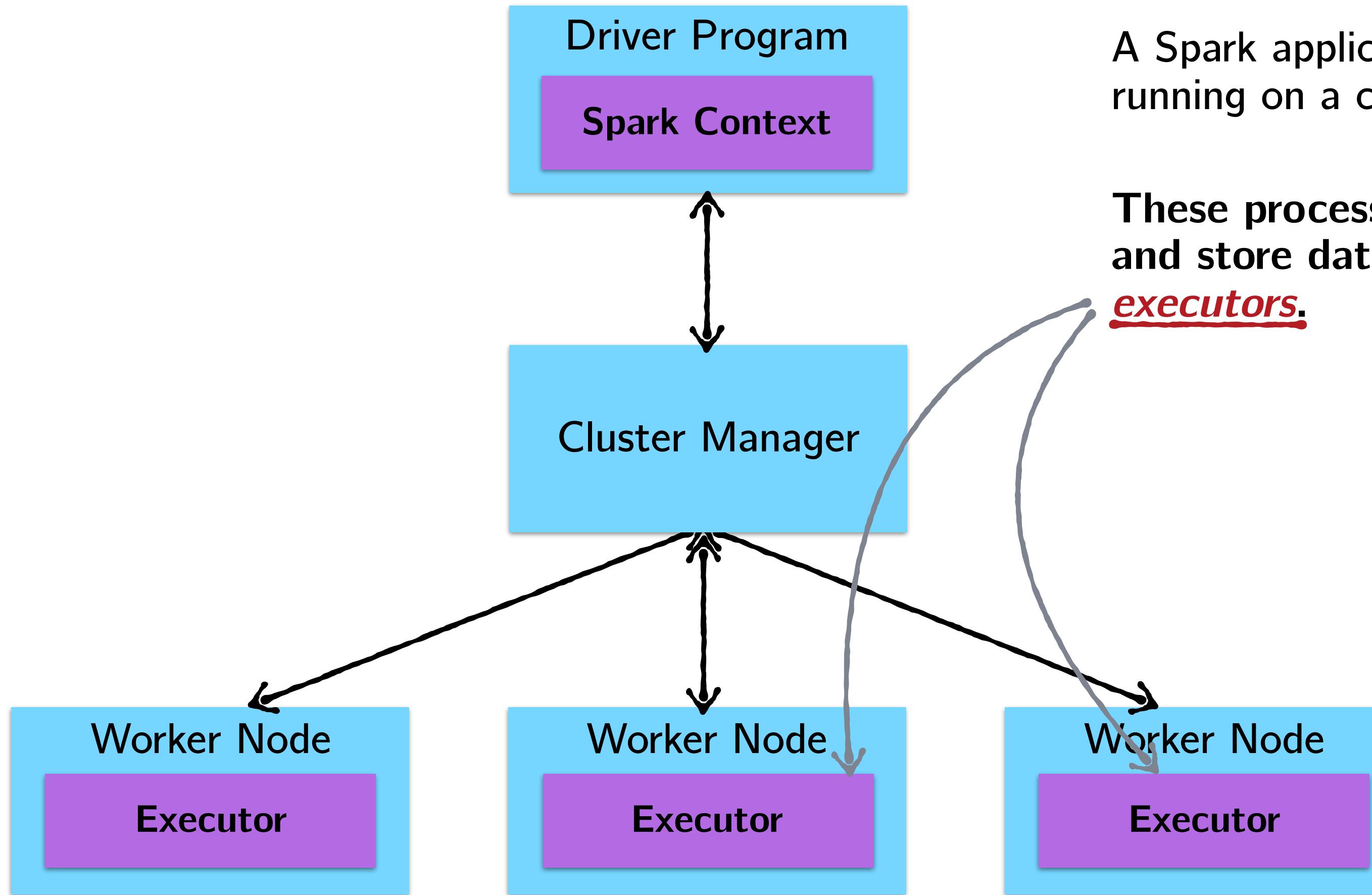
How Spark Jobs are Executed



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**.

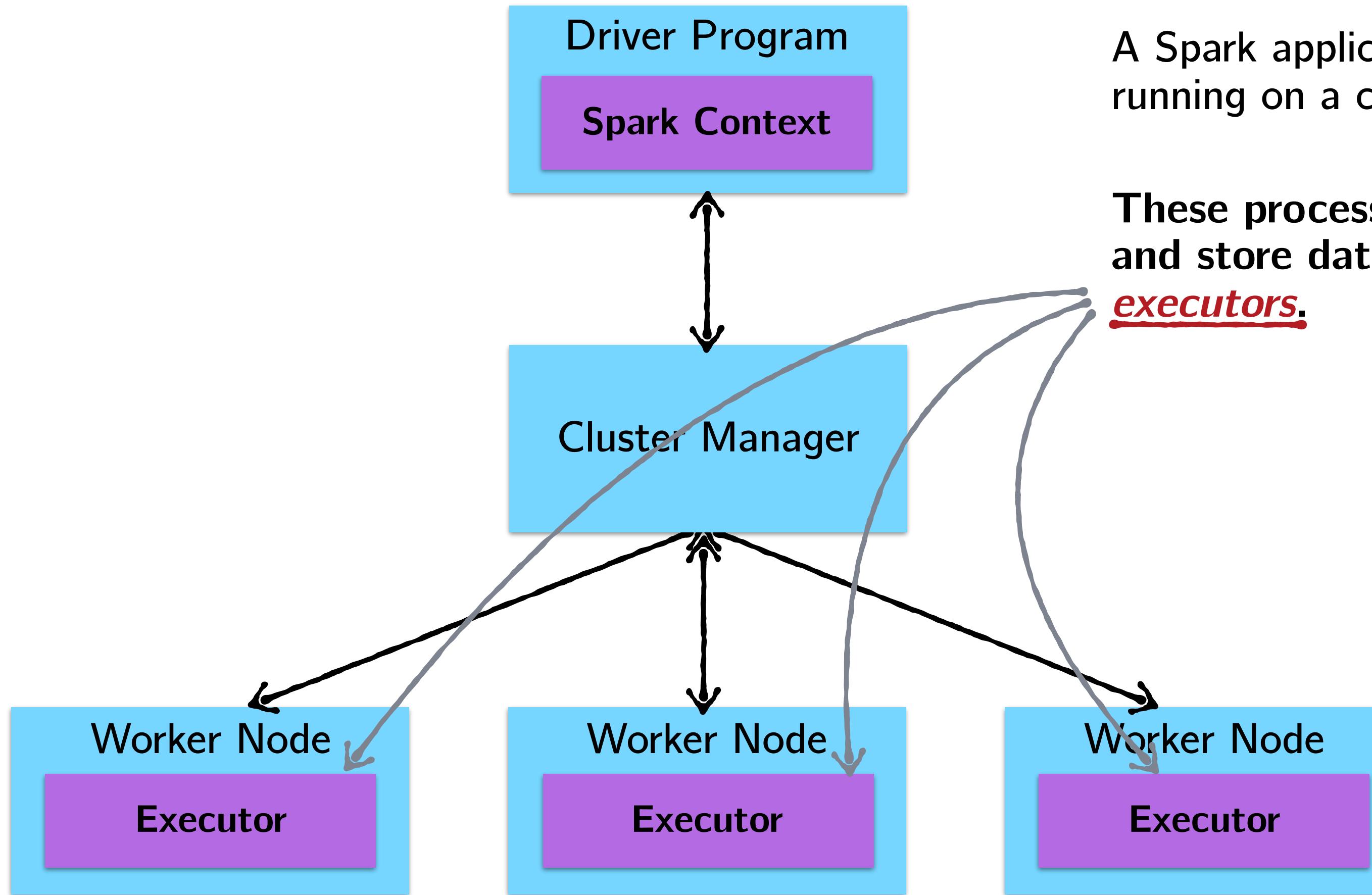
How Spark Jobs are Executed



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**.

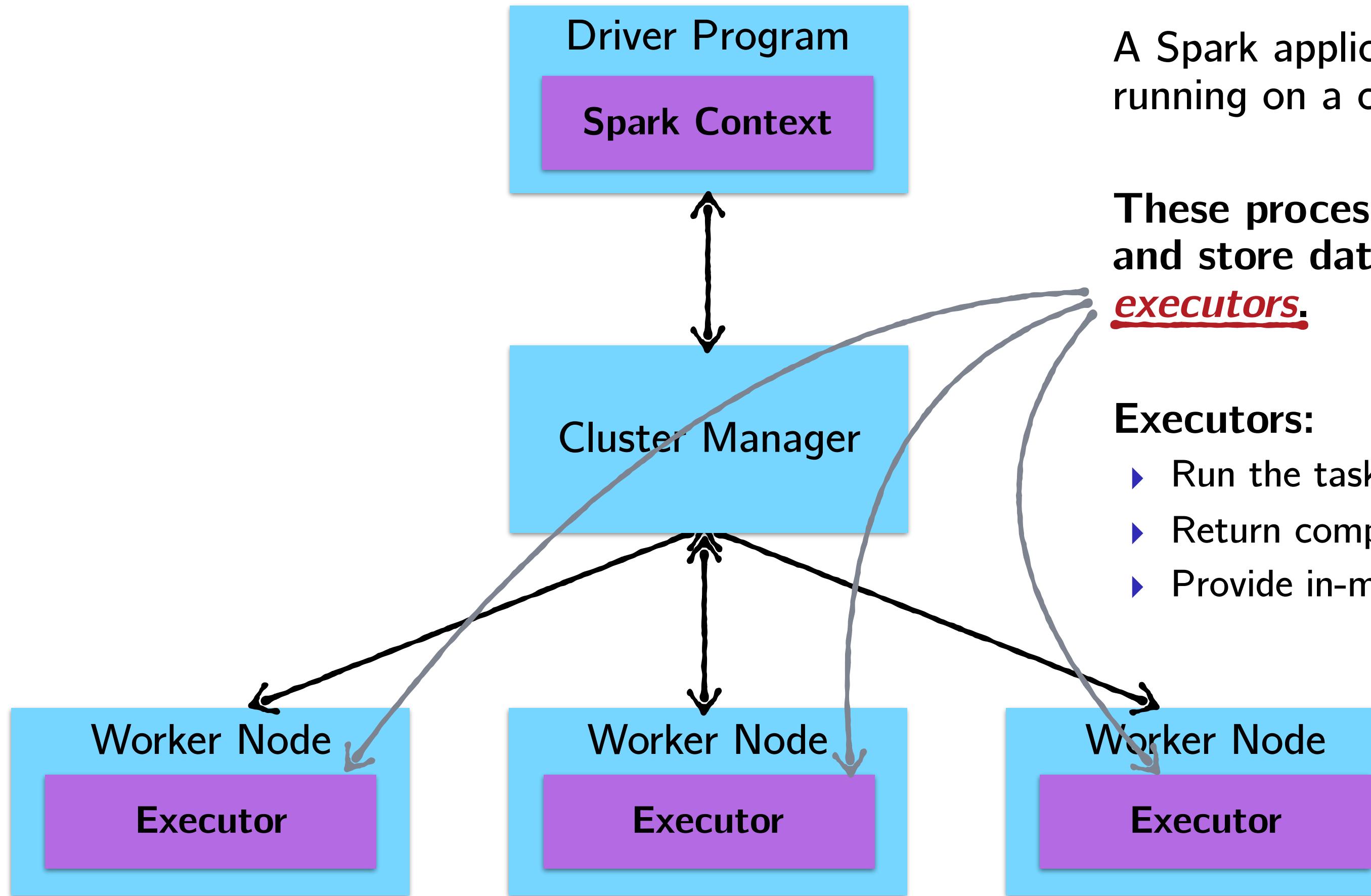
How Spark Jobs are Executed



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



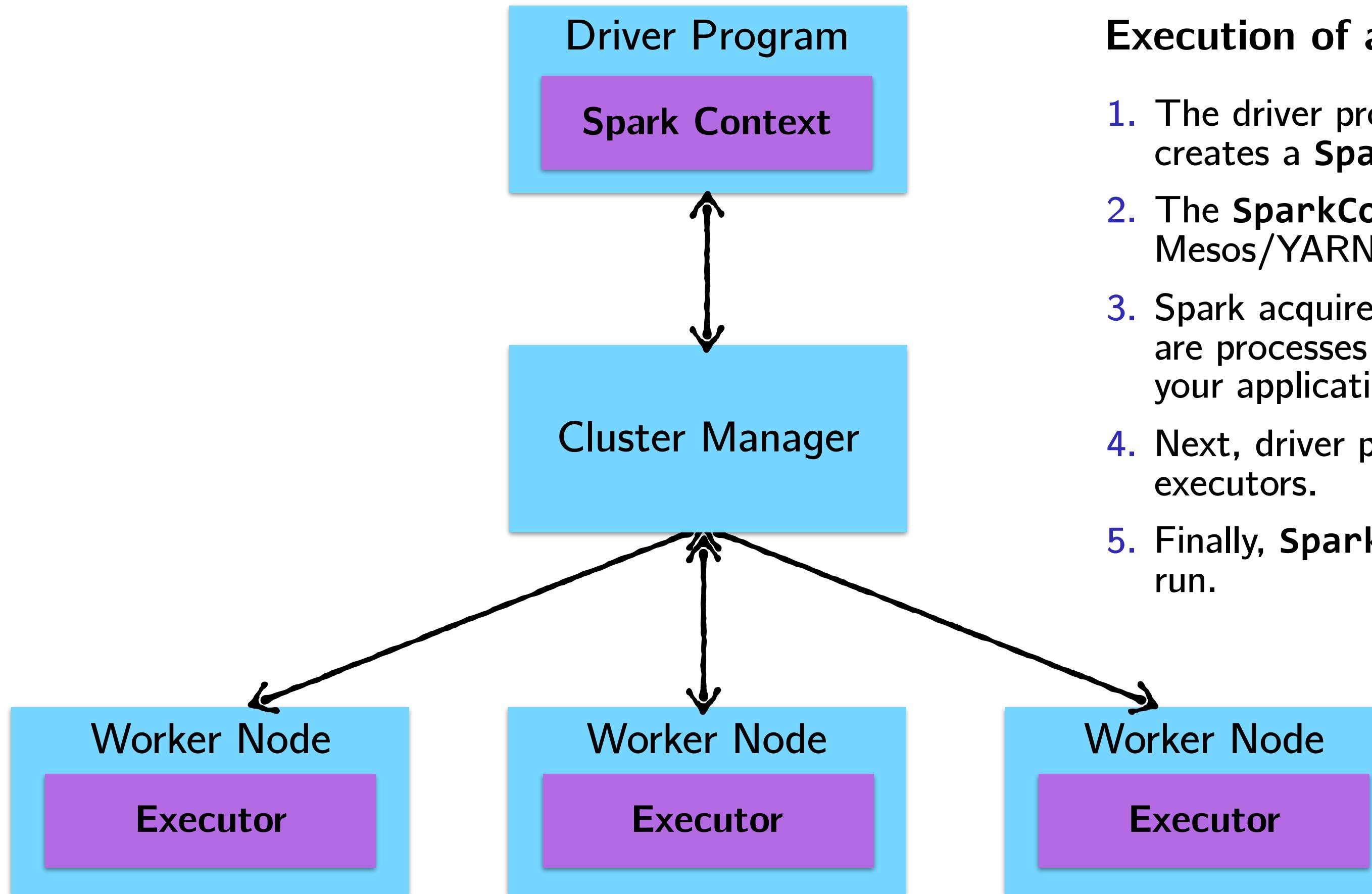
A Spark application is a set of processes running on a cluster.

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:

```
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?

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 to 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.



ÉCOLE POLYTECHNIQUE
FÉDÉRALE DE LAUSANNE

Reduction Operations

Big Data Analysis with Scala and Spark

Heather Miller

What we've seen so far

- ▶ we defined *Distributed Data Parallelism*
- ▶ we saw that Apache Spark implements this model
- ▶ we got a feel for what latency means to distributed systems

What we've seen so far

- ▶ we defined *Distributed Data Parallelism*
- ▶ we saw that Apache Spark implements this model
- ▶ we got a feel for what latency means to distributed systems

Spark's Programming Model

- ▶ We saw that, at a glance, Spark looks like Scala collections
- ▶ However, internally, Spark behaves differently than Scala collections
 - ▶ Spark uses *laziness* to save time and memory
- ▶ We saw *transformations* and *actions*
- ▶ We saw caching and persistence (*i.e.*, cache in memory, save time!)
- ▶ We saw how the cluster topology comes into the programming model

Transformations to Actions

Most of our intuitions have focused on distributing **transformations** such as map, flatMap, filter, etc.

We've visualized how transformations like these are distributed and parallelized.

Transformations to Actions

Most of our intuitions have focused on distributing **transformations** such as map, flatMap, filter, etc.

We've visualized how transformations like these are distributed and parallelized.

But what about actions? In particular, how are common reduce-like actions distributed in Spark?

Reduction Operations, Generally

First, what do we mean by “reduction operations”?

Recall operations such as fold, reduce, and aggregate from Scala sequential collections. All of these operations and their variants (such as foldLeft, reduceRight, etc) have something in common.

Reduction Operations, Generally

First, what do we mean by “reduction operations”?

Recall operations such as fold, reduce, and aggregate from Scala sequential collections. All of these operations and their variants (such as foldLeft, reduceRight, etc) have something in common.

Reduction Operations:

walk though a collection and combine neigboring elements of the collection together to produce a single combined result.

(rather than another collection)

Reduction Operations, Generally

Reduction Operations:

walk though a collection and combine neighboring elements of the collection together to produce a single combined result.
(rather than another collection)

Example:

```
case class Taco(kind: String, price: Double)
```

```
val tacoOrder =  
  List(  
    Taco("Carnitas", 2.25),  
    Taco("Corn", 1.75),  
    Taco("Barbacoa", 2.50),  
    Taco("Chicken", 2.00))
```

```
val cost = tacoOrder.foldLeft(0.0)((sum, taco) => sum + taco.price)
```

Parallel Reduction Operations

Recall what we learned in the course Parallel Programming course about foldLeft vs fold.

Which of these two were parallelizable?

Parallel Reduction Operations

Recall what we learned in the course Parallel Programming course about `foldLeft` vs `fold`.

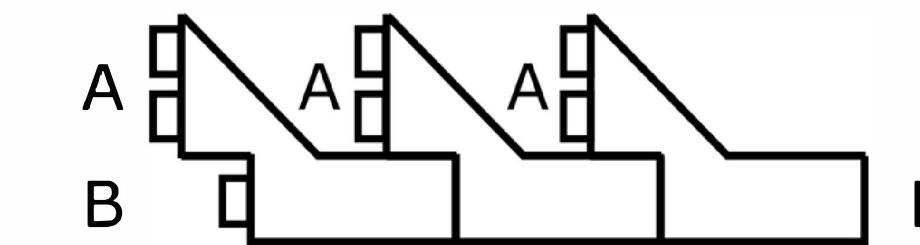
Which of these two were parallelizable?

foldLeft is not parallelizable.

```
def foldLeft[B](z: B)(f: (B, A) => B): B
```

Applies a binary operator to a start value and all elements of this collection or iterator, going left to right.

— Scala API documentation



Parallel Reduction Operations: FoldLeft

foldLeft is not parallelizable.

```
def foldLeft[B](z: B)(f: (B, A) => B): B
```

Being able to change the result type from A to B forces us to have to execute foldLeft sequentially from left to right.

Concretely, given:

`"1234"`

```
val xs = List(1, 2, 3, 4)  
val res = xs.foldLeft("")((str: String, i: Int) => str + i)
```

What happens if we try to break this collection in two and parallelize?

Parallel Reduction Operations: FoldLeft

foldLeft is not parallelizable.

```
def foldLeft[B](z: B)(f: (B, A) => B): B
```

```
val xs = List(1, 2, 3, 4)  
val res = xs.foldLeft("")((str: String, i: Int) => str + i) String
```

List(1,2)
 $\frac{}{'''+1 \rightarrow "1"}\quad$
 $\frac{}{"1"+2 \rightarrow "12"}\quad$
string

List(3,4)
 $\frac{}{"'''+3 \rightarrow "3"}\quad$
 $\frac{}{"3"+4 \rightarrow "34"}\quad$
String

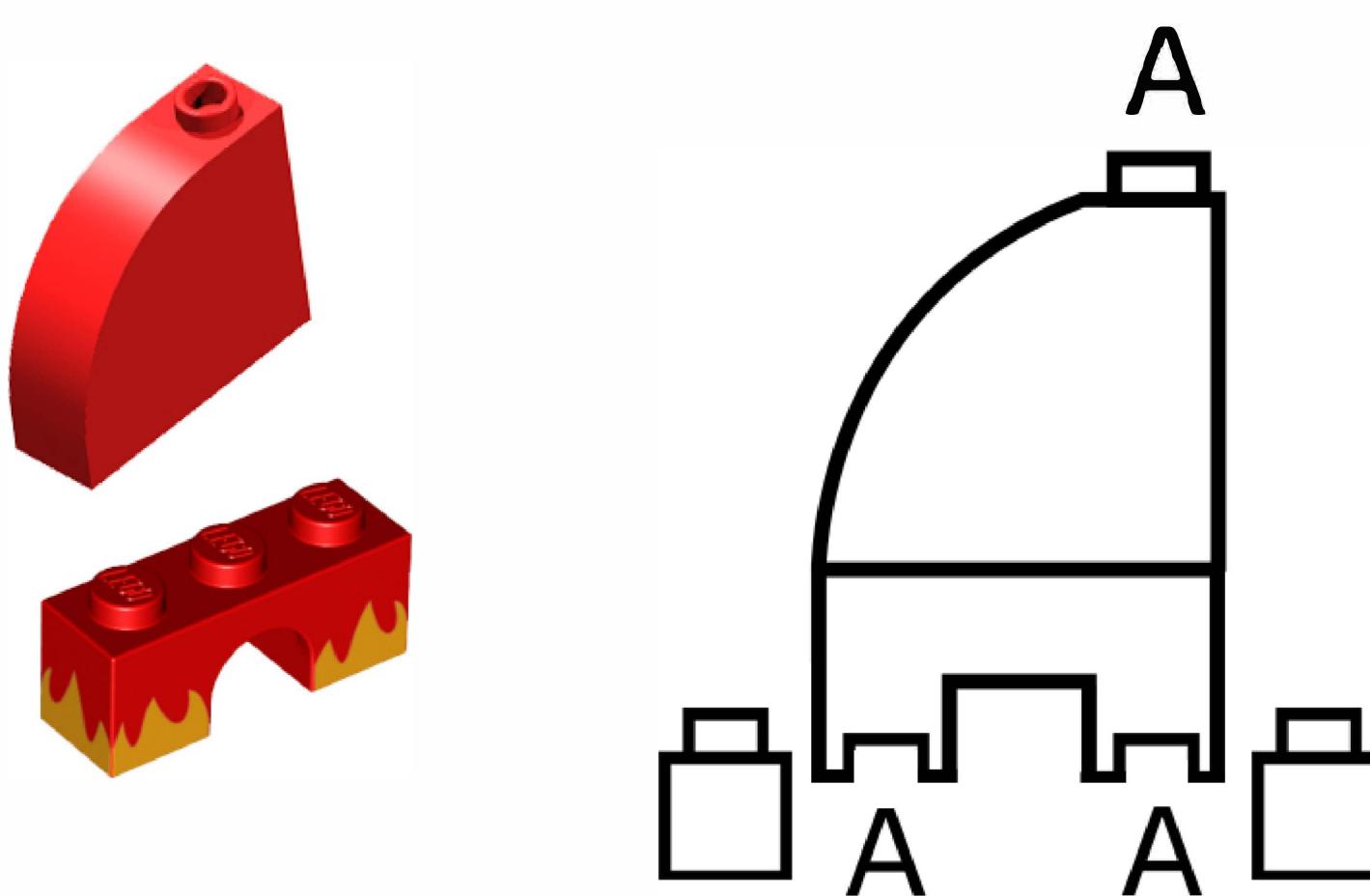
!! type error !!

can't apply
 $(str: String, i: Int) \rightarrow str + i$!!

Parallel Reduction Operations: Fold

fold enables us to parallelize things, but it restricts us to always returning the same type.

```
def fold(z: A)(f: (A, A) => A): A
```

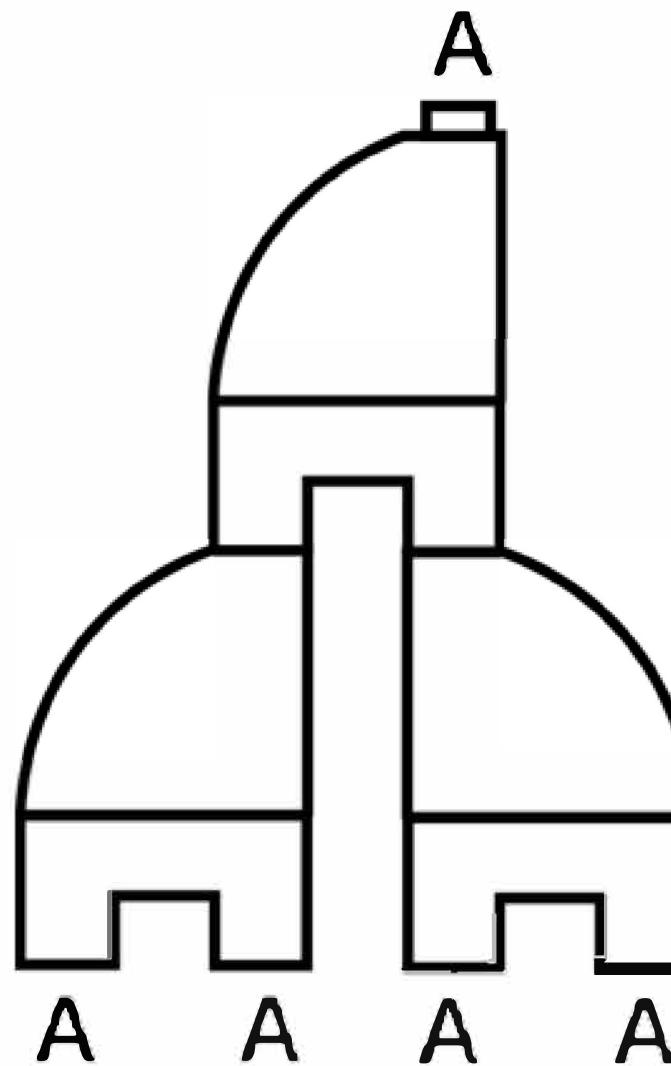
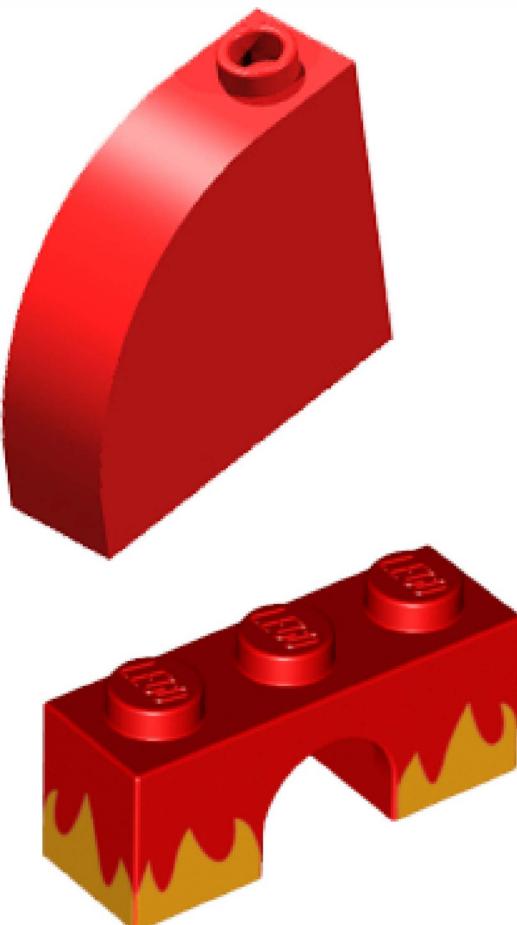


It enables us to parallelize using a single function f by enabling us to build parallelizable reduce trees.

Parallel Reduction Operations: Fold

It enables us to parallelize using a single function f by enabling us to build parallelizable reduce trees.

```
def fold(z: A)(f: (A, A) => A): A
```



Parallel Reduction Operations: Aggregate

Does anyone remember what aggregate does?

Parallel Reduction Operations: Aggregate

Does anyone remember what aggregate does?

```
aggregate[B](z: => B)(seqop: (B, A) => B, combop: (B, B) => B): B
```

Parallel Reduction Operations: Aggregate

Does anyone remember what aggregate does?

```
aggregate[B](z: => B)(seqop: (B, A) => B, combop: (B, B) => B): B
```

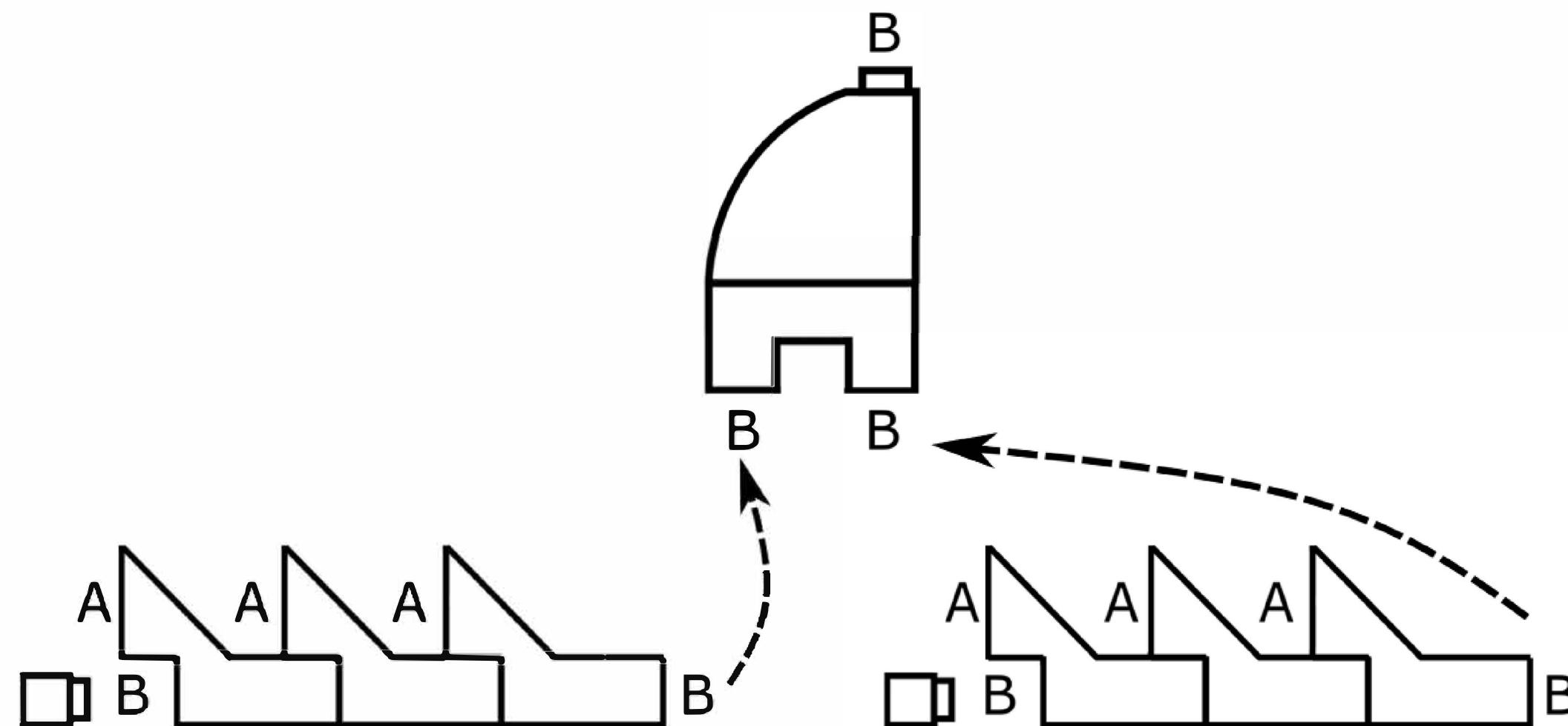
aggregate is said to be general because it gets you the best of both worlds.

Properties of aggregate

1. Parallelizable.
2. Possible to change the return type.

Parallel Reduction Operations: Aggregate

aggregate[B](z: => B)(seqop: (B, A) => B, combop: (B, B) => B): B



Aggregate lets you still do sequential-style folds *in chunks* which change the result type. Additionally requiring the combop function enables building one of these nice reduce trees that we saw is possible with fold to *combine these chunks* in parallel.

Reduction Operations on RDDs

Scala collections:

`fold`

`foldLeft/foldRight`

`reduce`

`aggregate`

Spark:

`fold`

~~`foldLeft/foldRight`~~

`reduce`

`aggregate`

Reduction Operations on RDDs

Scala collections:

`fold`

`foldLeft/foldRight`

`reduce`

`aggregate`

Spark:

`fold`

~~`foldLeft/foldRight`~~

`reduce`

`aggregate`

Spark doesn't even give you the option to use `foldLeft/foldRight`. Which means that if you have to change the return type of your reduction operation, your only choice is to use `aggregate`.

Reduction Operations on RDDs

Scala collections:

fold

foldLeft/foldRight

reduce

aggregate

Spark:

fold

~~foldLeft/foldRight~~

reduce

aggregate

Spark doesn't even give you the option to use foldLeft/foldRight. Which means that if you have to change the return type of your reduction operation, your only choice is to use aggregate.

Question: Why not still have a serial foldLeft/foldRight on Spark?

Reduction Operations on RDDs

Scala collections:

`fold`

`foldLeft/foldRight`

`reduce`

`aggregate`

Spark:

`fold`

~~`foldLeft/foldRight`~~

`reduce`

`aggregate`

Spark doesn't even give you the option to use `foldLeft/foldRight`. Which means that if you have to change the return type of your reduction operation, your only choice is to use `aggregate`.

Question: Why not still have a serial `foldLeft/foldRight` on Spark?

Doing things serially across a cluster is actually difficult. Lots of synchronization. Doesn't make a lot of sense.

RDD Reduction Operations: Aggregate

In Spark, aggregate is a more desirable reduction operator a majority of the time. Why do you think that's the case?

RDD Reduction Operations: Aggregate

In Spark, aggregate is a more desirable reduction operator a majority of the time. Why do you think that's the case?

As you will realize from experimenting with our Spark ~~cluster~~^{assignments}, much of the time when working with large-scale data, our goal is to ***project down from larger/more complex data types.***

RDD Reduction Operations: Aggregate

In Spark, aggregate is a more desirable reduction operator a majority of the time. Why do you think that's the case?

As you will realize from experimenting with our Spark cluster, much of the time when working with large-scale data, our goal is to ***project down from larger/more complex data types.***

Example:

```
case class WikipediaPage(  
    title: String,  
    redirectTitle: String,  
    timestamp: String,  
    lastContributorUsername: String,  
    text: String)
```



RDD Reduction Operations: Aggregate

As you will realize after experimenting with Spark a bit, much of the time when working with large-scale data, your goal is to *project down from larger/more complex data types*.

Example:

```
case class WikipediaPage(  
    title: String,  
    redirectTitle: String,  
    timestamp: String,  
    lastContributorUsername: String,  
    text: String)
```

I might only care about title and timestamp, for example. In this case, it'd save a lot of time/memory to not have to carry around the full-text of each article (text) in our accumulator!

Hence, why accumulate is often more desirable in Spark than in Scala collections!



ÉCOLE POLYTECHNIQUE
FÉDÉRALE DE LAUSANNE

Distributed Key-Value Pairs (Pair RDDs)

Big Data Analysis with Scala and Spark

Heather Miller

Distributed Key-Value Pairs

In single-node Scala, key-value pairs can be thought of as *maps*.
(Or *associative arrays* or *dictionaries* in JavaScript or Python)

Distributed Key-Value Pairs

In single-node Scala, key-value pairs can be thought of as *maps*.
(Or *associative arrays* or *dictionaries* in JavaScript or Python)

While maps/dictionaries/etc are available across most languages, they perhaps aren't the most commonly-used structure in single-node programs.
List/Arrays probably more common.

**Most common in world of big data processing:
Operating on data in the form of key-value pairs.**

- ▶ Manipulating key-value pairs a key choice in design of MapReduce

Distributed Key-Value Pairs

MapReduce: Simplified Data Processing on Large Clusters

Jeffrey Dean and Sanjay Ghemawat

jeff@google.com, sanjay@google.com

(2004 research paper)

Google, Inc.

Abstract

MapReduce is a programming model and an associated implementation for processing and generating large data sets. Users specify a *map* function that processes a key/value pair to generate a set of intermediate key/value pairs, and a *reduce* function that merges all intermediate values associated with the same intermediate key. Many real world tasks are expressible in this model, as shown in the paper.

Programs written in this functional style are automatically parallelized and executed on a large cluster of commodity machines. The run-time system takes care of the details of partitioning the input data, scheduling the pro-

given day, etc. Most such computations are conceptually straightforward. However, the input data is usually large and the computations have to be distributed across hundreds or thousands of machines in order to finish in a reasonable amount of time. The issues of how to parallelize the computation, distribute the data, and handle failures conspire to obscure the original simple computation with large amounts of complex code to deal with these issues.

As a reaction to this complexity, we designed a new abstraction that allows us to express the simple computations we were trying to perform but hides the messy details of parallelization, fault-tolerance, data distribution

Distributed Key-Value Pairs

Abstract

MapReduce is a programming model and an associated implementation for processing and generating large data sets. Users specify a *map* function that processes a key/value pair to generate a set of intermediate key/value pairs, and a *reduce* function that merges all intermediate values associated with the same intermediate key. Many real world tasks are expressible in this model, as shown in the paper.

(2004 research paper)

Programs written in this functional style are automatically parallelized and executed on a large cluster of commodity machines. The run-time system takes care of the details of partitioning the input data, scheduling the program's execution across a set of machines, handling machine failures, and managing the required inter-machine communication. This allows programmers without any experience with parallel and distributed systems to easily utilize the resources of a large distributed system.

Our implementation of MapReduce runs on a large cluster of commodity machines and is highly scalable: a typical MapReduce computation processes many terabytes of data on thousands of machines. Programmers find the system easy to use: hundreds of MapReduce programs have been implemented and upwards of one thousand MapReduce jobs are executed on Google's clusters every day.

given day, etc. Most such computations are conceptually straightforward. However, the input data is usually large and the computations have to be distributed across hundreds or thousands of machines in order to finish in a reasonable amount of time. The issues of how to parallelize the computation, distribute the data, and handle failures conspire to obscure the original simple computation with large amounts of complex code to deal with these issues.

As a reaction to this complexity, we designed a new abstraction that allows us to express the simple computations we were trying to perform but hides the messy details of parallelization, fault-tolerance, data distribution and load balancing in a library. Our abstraction is inspired by the *map* and *reduce* primitives present in Lisp and many other functional languages. We realized that most of our computations involved applying a *map* operation to each logical “record” in our input in order to compute a set of intermediate key/value pairs, and then applying a *reduce* operation to all the values that shared the same key, in order to combine the derived data appropriately. Our use of a functional model with user-specified map and reduce operations allows us to parallelize large computations easily and to use re-execution as the primary mechanism for fault tolerance.

The major contributions of this work are a simple and powerful interface that enables automatic parallelization and distribution of large-scale computations, combined

Distributed Key-Value Pairs (Pair RDDs)

Large datasets are often made up of unfathomably large numbers of complex, nested data records.

To be able to work with such datasets, it's often desirable to *project down* these complex datatypes into **key-value pairs**.

Distributed Key-Value Pairs (Pair RDDs)

```
{  
  "definitions": {  
    "firstname": "string",  
    "lastname": "string",  
    "address": {  
      "type": "object",  
      "properties": {  
        "street_address": {  
          "type": "string"  
        },  
        "city": {  
          "type": "string"  
        },  
        "state": {  
          "type": "string"  
        }  
      },  
      "required": [  
        "street_address",  
        "city",  
        "state"  
      ]  
    }  
  }  
}
```

Large datasets are often made up of unfathomably large numbers of complex, nested data records.

To be able to work with such datasets, it's often desirable to *project down* these complex datatypes into **key-value pairs**.

Example:

In the JSON record to the left, it may be desirable to create an RDD of properties of type:

```
RDD[(String, Property)] // where 'String' is a key representing a city,  
                         // and 'Property' is its corresponding value.
```

```
case class Property(street: String, city: String, state: String)
```

where instances of Properties can be grouped by their respective cities and represented in a RDD of key-value pairs.

Distributed Key-Value Pairs (Pair RDDs)

Often when working with distributed data, it's useful to organize data into **key-value pairs**.

In Spark, distributed key-value pairs are “Pair RDDs.”

Useful because: Pair RDDs allow you to act on each key in parallel or regroup data across the network.

Distributed Key-Value Pairs (Pair RDDs)

Often when working with distributed data, it's useful to organize data into **key-value pairs**.

In Spark, distributed key-value pairs are “Pair RDDs.”

Useful because: Pair RDDs allow you to act on each key in parallel or regroup data across the network.

Pair RDDs have additional, specialized methods for working with data associated with keys. RDDs are parameterized by a pair are Pair RDDs.

`RDD[(K,V)] // <== treated specially by Spark!`

Pair RDDs (Key-Value Pairs)

Key-value pairs are known as Pair RDDs in Spark.

When an RDD is created with a pair as its element type, Spark automatically adds a number of extra useful additional methods (extension methods) for such pairs.

Some of the most important extension methods for RDDs containing pairs (e.g., `RDD[(K, V)]`) are:

```
def groupByKey(): RDD[(K, Iterable[V])]  
def reduceByKey(func: (V, V) => V): RDD[(K, V)]  
def join[W](other: RDD[(K, W)]): RDD[(K, (V, W))]
```

Pair RDDs (Key-Value Pairs)

Creating a Pair RDD

Pair RDDs are most often created from already-existing non-pair RDDs, for example by using the `map` operation on RDDs:

```
val rdd: RDD[WikipediaPage] = ...
```

```
val pairRdd = ???
```

Pair RDDs (Key-Value Pairs)

Creating a Pair RDD

Pair RDDs are most often created from already-existing non-pair RDDs, for example by using the `map` operation on RDDs:

```
val rdd: RDD[WikipediaPage] = ...
```

```
// Has type: org.apache.spark.rdd.RDD[(String, String)]  
val pairRdd = rdd.map(page => (page.title, page.text))
```

Once created, you can now use transformations specific to key-value pairs such as `reduceByKey`, `groupByKey`, and `join`



ÉCOLE POLYTECHNIQUE
FÉDÉRALE DE LAUSANNE

Transformations and Actions on Pair RDDs

Big Data Analysis with Scala and Spark

Heather Miller

Some interesting Pair RDDs operations

Important operations defined on Pair RDDs:
(But not available on regular RDDs)

Transformations

- ▶ groupByKey
- ▶ reduceByKey
- ▶ mapValues
- ▶ keys
- ▶ join
- ▶ leftOuterJoin/rightOuterJoin

Action

- ▶ countByKey

Pair RDD Transformation: groupByKey

Recall groupBy from Scala collections.

Pair RDD Transformation: ~~groupByKey~~

Recall `groupBy` from Scala collections.

```
def groupBy[K](f: A => K): Map[K, Traversable[A]]
```

Partitions this traversable collection into a map of traversable collections according to some discriminator function.

In English: Breaks up a collection into two or more collections according to a function that you pass to it. Result of the function is the key, the collection of results that return that key when the function is applied to it. Returns a Map mapping computed keys to collections of corresponding values.

Pair RDD Transformation: groupByKey

Recall groupBy from Scala collections.

```
def groupBy[K](f: A => K): Map[K, Traversable[A]]
```

Example:

Let's group the below list of ages into "child", "adult", and "senior" categories.

```
val ages = List(2, 52, 44, 23, 17, 14, 12, 82, 51, 64)
val grouped = ages.groupBy { age =>
    if (age >= 18 && age < 65) "adult"
    else if (age < 18) "child"
    else "senior"
}
// grouped: scala.collection.immutable.Map[String,List[Int]] =
// Map(senior -> List(82), adult -> List(52, 44, 23, 51, 64),
// child -> List(2, 17, 14, 12))
```

Pair RDD Transformation: groupByKey

Recall groupBy from Scala collections. groupByKey can be thought of as a groupBy on Pair RDDs that is specialized on grouping all values that have the same key. As a result, it takes no argument.

```
def groupByKey(): RDD[(K, Iterable[V])]
```

Pair RDD Transformation: groupByKey

Recall groupBy from Scala collections. groupByKey can be thought of as a groupBy on Pair RDDs that is specialized on grouping all values that have the same key. As a result, it takes no argument.

```
def groupByKey(): RDD[(K, Iterable[V])]
```

Example:

```
case class Event(organizer: String, name: String, budget: Int)
```

```
val eventsRdd = sc.parallelize(...)  
    .map(event => (event.organizer, event.budget))
```

```
val groupedRdd = eventsRdd.groupByKey()
```

Here the key is organizer. What does this call do?

Pair RDD Transformation: groupByKey

Example:

```
case class Event(organizer: String, name: String, budget: Int)
val eventsRdd = sc.parallelize(...)
    .map(event => (event.organizer, event.budget))

val groupedRdd = eventsRdd.groupByKey()

// TRICK QUESTION! As-is, it "does" nothing. It returns an unevaluated RDD

groupedRdd.collect().foreach(println)
// (Prime Sound,CompactBuffer(42000))
// (Sportorg,CompactBuffer(23000, 12000, 1400))
// ...
```

Pair RDD Transformation: reduceByKey

Conceptually, `reduceByKey` can be thought of as a combination of `groupByKey` and `reduce-ing` on all the values per key. It's more efficient though, than using each separately. (We'll see why later.)

```
def reduceByKey(func: (V, V) => V): RDD[(K, V)]
```

Pair RDD Transformation: reduceByKey

Conceptually, reduceByKey can be thought of as a combination of groupByKey and reduce-ing on all the values per key. It's more efficient though, than using each separately. (We'll see why later.)

```
def reduceByKey(func: (V, V) => V): RDD[(K, V)]
```

Example: Let's use eventsRdd from the previous example to calculate the total budget per organizer of all of their organized events.

```
case class Event(organizer: String, name: String, budget: Int)
val eventsRdd = sc.parallelize(...)
    .map(event => (event.organizer, event.budget))

val budgetsRdd = ...
```

Pair RDD Transformation: reduceByKey

Example: Let's use eventsRdd from the previous example to calculate the total budget per organizer of all of their organized events.

```
case class Event(organizer: String, name: String, budget: Int)  
val eventsRdd = sc.parallelize(...)  
    .map(event => (event.organizer, event.budget))
```

```
val budgetsRdd = eventsRdd.reduceByKey(_+_)
```

```
reducedRdd.collect().foreach(println)  
// (Prime Sound,42000)  
// (Sportorg,36400)  
// (Innotech,320000)  
// (Association Balélec,50000)
```

Pair RDD Transformation: `mapValues` and Action: `countByKey`

`mapValues` (`def mapValues[U](f: V => U): RDD[(K, U)]`) can be thought of as a short-hand for:

```
rdd.map { case (x, y): (x, func(y))}
```

That is, it simply applies a function to only the values in a Pair RDD.

`countByKey` (`def countByKey(): Map[K, Long]`) simply counts the number of elements per key in a Pair RDD, returning a normal Scala Map (remember, it's an action!) mapping from keys to counts.

Pair RDD Transformation: mapValues and Action: countByKey

Example: we can use each of these operations to compute the average budget per event organizer, if possible.

```
// Calculate a pair (as a key's value) containing (budget, #events)  
val intermediate = ??? // Can we use countByKey?
```

Pair RDD Transformation: mapValues and Action: countByKey

Example: we can use each of these operations to compute the average budget per event organizer, if possible.

```
// Calculate a pair (as a key's value) containing (budget, #events)
```

```
val intermediate =
```

```
eventsRdd.mapValues(b => (b, 1))
```

```
.reduceByKey(
```

$$\begin{array}{c} (\text{org}, \text{budget}) \\ K \qquad V \end{array} \rightarrow \underline{\text{org}} \underline{(\text{budget}, 1)}$$

Result should look like:

$(\text{org}, (\text{total Budget}, \text{total \# events organized}))$

Pair RDD Transformation: mapValues and Action: countByKey

Example: we can use each of these operations to compute the average budget per event organizer, if possible.

```
// Calculate a pair (as a key's value) containing (budget, #events)
val intermediate =
  eventsRdd.mapValues(b => (b, 1))
    .reduceByKey((v1, v2) => (v1._1 + v2._1, v1._2 + v2._2))
// intermediate: RDD[(String, (Int, Int))]
```

The diagram illustrates the transformation of an RDD. It starts with a single column of values. After applying `mapValues(b => (b, 1))`, it becomes a two-column RDD where the first column is labeled "budgets" and the second column is labeled "total # events". Finally, after applying `reduceByKey((v1, v2) => (v1._1 + v2._1, v1._2 + v2._2))`, it becomes a single column of values again.

Pair RDD Transformation: mapValues and Action: countByKey

Example: we can use each of these operations to compute the average budget per event organizer, if possible.

```
// Calculate a pair (as a key's value) containing (budget, #events)
val intermediate =
  eventsRdd.mapValues(b => (b, 1))
    .reduceByKey((v1, v2) => (v1._1 + v2._1, v1._2 + v2._2))
// intermediate: RDD[(String, (Int, Int))]

val avgBudgets = ???
```

Pair RDD Transformation: mapValues and Action: countByKey

Example: we can use each of these operations to compute the average budget per event organizer, if possible.

```
// Calculate a pair (as a key's value) containing (budget, #events)
val intermediate =
  eventsRdd.mapValues(b => (b, 1))
    .reduceByKey((v1, v2) => (v1._1 + v2._1, v1._2 + v2._2))
// intermediate: RDD[(String, (Int, Int))]

val avgBudgets = intermediate.mapValues {
  case (budget, numberEvents) => budget / numberEvents
}
avgBudgets.collect().foreach(println)
// (Prime Sound,42000)
// (Sportorg,12133)
// (Innotech,106666)
// (Association Balélec,50000)
```

Pair RDD Transformation: keys

keys (def keys: RDD[K]) Return an RDD with the keys of each tuple.

Note: this method is a transformation and thus returns an RDD because the number of keys in a Pair RDD may be unbounded. It's possible for every value to have a unique key, and thus it may not be possible to collect all keys at one node.

Pair RDD Transformation: keys

keys (def keys: RDD[K]) Return an RDD with the keys of each tuple.

Note: this method is a transformation and thus returns an RDD because the number of keys in a Pair RDD may be unbounded. It's possible for every value to have a unique key, and thus it may not be possible to collect all keys at one node.

Example: we can count the number of unique visitors to a website using the keys transformation.

```
case class Visitor(ip: String, timestamp: String, duration: String)
val visits: RDD[Visitor] = sc.textfile(...)
  .map(v => (v.ip, v.duration))

val numUniqueVisits = ???
```

Pair RDD Transformation: keys

keys (def keys: RDD[K]) Return an RDD with the keys of each tuple.

Note: this method is a transformation and thus returns an RDD because the number of keys in a Pair RDD may be unbounded. It's possible for every value to have a unique key, and thus it may not be possible to collect all keys at one node.

Example: we can count the number of unique visitors to a website using the keys transformation.

```
case class Visitor(ip: String, timestamp: String, duration: String)
val visits: RDD[Visitor] = sc.textfile(...)
    .map( v => (v.ip, v.duration))
val numUniqueVisits = visits.keys.distinct().count()
// numUniqueVisits: Long = 3391
```

PairRDDFunctions

For a list of all available specialized Pair RDD operations, see the Spark API page for `PairRDDFunctions` (ScalaDoc):

<http://spark.apache.org/docs/latest/api/scala/index.html#org.apache.spark.rdd.PairRDDFunctions>

The screenshot shows the ScalaDoc interface for the `PairRDDFunctions` class. At the top, there's a navigation bar with a logo, the package name `org.apache.spark.rdd`, the class name `PairRDDFunctions`, and a link to the `Related Doc: package rdd`. Below the header, the class definition is shown: `class PairRDDFunctions[K, V] extends Logging with Serializable`. A note below it states: "Extra functions available on RDDs of (key, value) pairs through an implicit conversion." Navigation links include "Source" (leading to `PairRDDFunctions.scala`) and "Linear Supertypes". The main content area has a search bar at the top. Below it are filtering options: "Ordering" (set to "Alphabetic"), "Inherited" (listing `PairRDDFunctions`, `Serializable`, `Serializable`, `Logging`, `AnyRef`, `Any`), and buttons for "Hide All" and "Show All". There are also "Visibility" filters for "Public" and "All". A section titled "Instance Constructors" contains the constructor definition: `new PairRDDFunctions(self: RDD[(K, V)])(implicit kt: ClassTag[K], vt: ClassTag[V], ord: Ordering[K] = null)`. The "Value Members" section lists several methods, each with a brief description:

- `aggregateByKey[U](zeroValue: U)(seqOp: (U, V) ⇒ U, combOp: (U, U) ⇒ U)(implicit arg0: ClassTag[U]): RDD[(K, U)]`
Aggregate the values of each key, using given combine functions and a neutral "zero value".
- `aggregateByKey[U](zeroValue: U, numPartitions: Int)(seqOp: (U, V) ⇒ U, combOp: (U, U) ⇒ U)(implicit arg0: ClassTag[U]): RDD[(K, U)]`
Aggregate the values of each key, using given combine functions and a neutral "zero value".
- `aggregateByKey[U](zeroValue: U, partitions: Partitioner)(seqOp: (U, V) ⇒ U, combOp: (U, U) ⇒ U)(implicit arg0: ClassTag[U]): RDD[(K, U)]`
Aggregate the values of each key, using given combine functions and a neutral "zero value".



ÉCOLE POLYTECHNIQUE
FÉDÉRALE DE LAUSANNE

Joins

Big Data Analysis with Scala and Spark

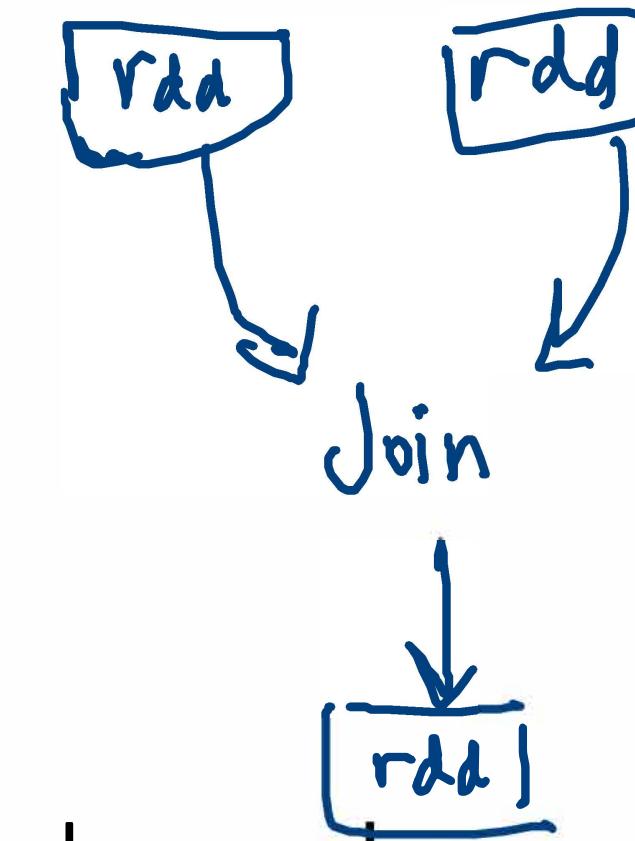
Heather Miller

Joins

Joins are another sort of transformation on Pair RDDs. They're used to combine multiple datasets. They are one of the most commonly-used operations on Pair RDDs!

There are two kinds of joins:

- ▶ Inner joins (join)
- ▶ Outer joins (leftOuterJoin/rightOuterJoin)



The key difference between the two is what happens to the keys when both RDDs don't contain the same key.

For example, if I were to join two RDDs containing different customer IDs (the key), the difference between inner/outer joins is what happens to customers whose IDs don't exist in both RDDs.

Example Dataset...

Example: Let's pretend the Swiss Rail company, CFF, has two datasets. One **RDD representing customers and their subscriptions (abos)**, and **another** representing customers and cities they frequently travel to (locations). **(E.g., gathered from CFF smartphone app.)*

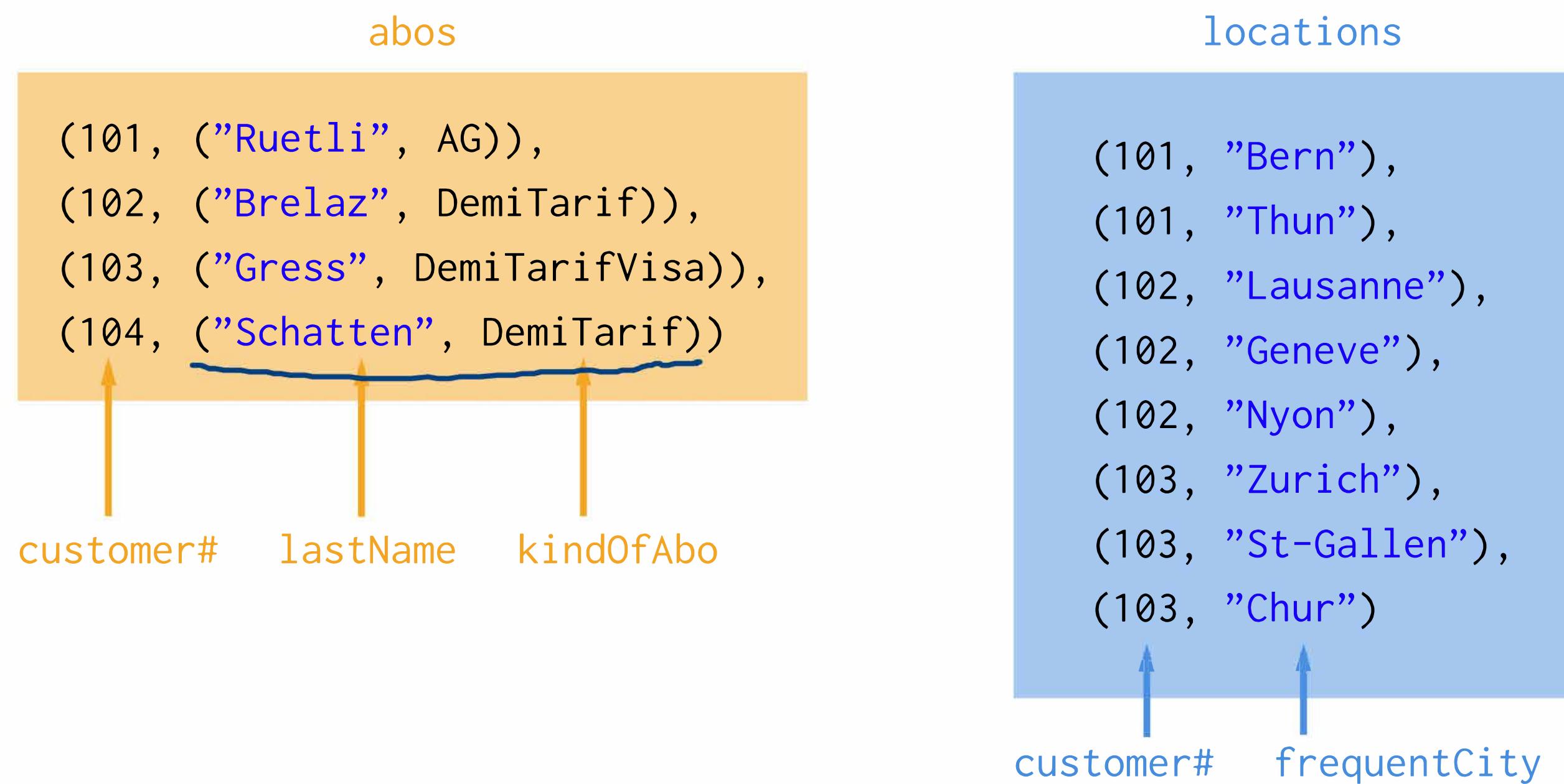
Let's assume the following concrete data:

```
val as = List((101, ("Ruetli", AG)), (102, ("Brelaz", DemiTarif)),  
              (103, ("Gress", DemiTarifVisa)), (104, ("Schatten", DemiTarif)))  
val abos = sc.parallelize(as)  
  
val ls = List((101, "Bern"), (101, "Thun"), (102, "Lausanne"), (102, "Geneve"),  
              (102, "Nyon"), (103, "Zurich"), (103, "St-Gallen"), (103, "Chur"))  
vals locations = sc.parallelize(ls)
```

Example Dataset... (2)

Example: Let's pretend the CFF has two datasets. One RDD representing customers and their subscriptions (abos), and another representing customers and cities they frequently travel to (locations). (*E.g.*, gathered from CFF smartphone app.)

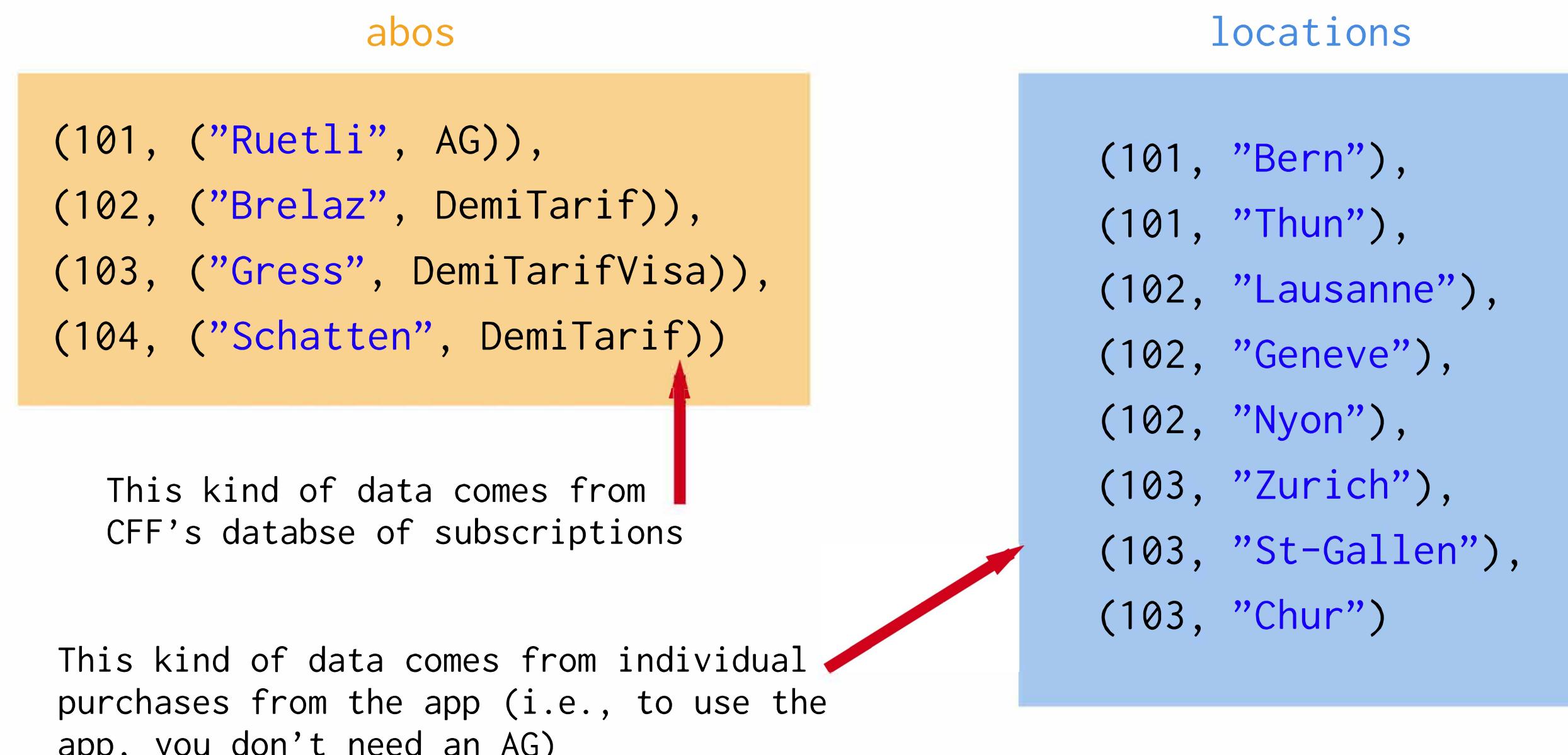
Let's assume the following concrete data: **(visualized)**



Example Dataset... (3)

Example: Let's pretend the CFF has two datasets. One RDD representing customers and their subscriptions (abos), and another representing customers and cities they frequently travel to (locations). (*E.g.*, gathered from CFF smartphone app.)

Let's assume the following concrete data: **(visualized)**



Inner Joins (join)

Inner joins return a new RDD containing combined pairs whose **keys are present in both input RDDs**.

```
def join[W](other: RDD[(K, W)]): RDD[(K, (V, W))]
```

Example: Let's pretend the CFF has two datasets. One RDD representing customers and their subscriptions (abos), and another representing customers and cities they frequently travel to (locations). (*E.g.*, gathered from CFF smartphone app.)

How do we combine only customers that have a subscription and where there is location info?

```
val abos = ... // RDD[Int, (String, Abonnement)]
val locations = ... // RDD[Int, String]
```

```
val trackedCustomers = ???
```

Inner Joins (join)

Example: Let's pretend the CFF has two datasets. One RDD representing customers and their subscriptions (abos), and another representing customers and cities they frequently travel to (locations). (*E.g.*, gathered from CFF smartphone app.)

How do we combine only customers that have a subscription and where there is location info?

```
val abos = ... // RDD[(Int, (String, Abonnement))]  
val locations = ... // RDD[(Int, String)]
```

Inner Joins (join)

Example continued with concrete data:

abos

```
(101, ("Ruetli", AG)),  
(102, ("Brelaz", DemiTarif)),  
(103, ("Gress", DemiTarifVisa)),  
(104, ("Schatten", DemiTarif))
```

locations

```
(101, "Bern"),  
(101, "Thun"),  
(102, "Lausanne"),  
(102, "Geneve"),  
(102, "Nyon"),  
(103, "Zurich"),  
(103, "St-Gallen"),  
(103, "Chur")
```

```
val trackedCustomers = abos.join(locations)  
// trackedCustomers: RDD[(Int, ((String, Abonnement), String))]
```

Inner Joins (join)

Example continued with concrete data:

abos

```
(101, ("Ruetli", AG)),  
(102, ("Brelaz", DemiTarif)),  
(103, ("Gress", DemiTarifVisa)),  
(104, ("Schatten", DemiTarif))
```

locations

```
(101, "Bern"),  
(101, "Thun"),  
(102, "Lausanne"),  
(102, "Geneve"),  
(102, "Nyon"),  
(103, "Zurich"),  
(103, "St-Gallen"),  
(103, "Chur")
```

We want to combine both RDDs into one:

How do we combine only customers that have a subscription and where there is location info?

Inner Joins (join)

Example continued with concrete data:

abos

```
(101, ("Ruetli", AG)),  
(102, ("Brelaz", DemiTarif)),  
(103, ("Gress", DemiTarifVisa)),  
(104, ("Schatten", DemiTarif))
```

locations

```
(101, "Bern"),  
(101, "Thun"),  
(102, "Lausanne"),  
(102, "Geneve"),  
(102, "Nyon"),  
(103, "Zurich"),  
(103, "St-Gallen"),  
(103, "Chur")
```

We want to make a new RDD with only these!

Inner Joins (join)

Example continued with concrete data:

trackedCustomers			
customer#	lastName	kindOfAbo	frequentCity
(101,((Ruetli,AG),Bern))			
(101,((Ruetli,AG),Thun))			
(102,((Brelaz,DemiTarif),Nyon))			
(102,((Brelaz,DemiTarif),Lausanne))			
(102,((Brelaz,DemiTarif),Geneve))			
(103,((Gress,DemiTarifVisa),St-Gallen))			
(103,((Gress,DemiTarifVisa),Chur))			
(103,((Gress,DemiTarifVisa),Zurich))			

```
val trackedCustomers = abos.join(locations)
// trackedCustomers: RDD[(Int, ((String, Abonnement), String))]
```

Inner Joins (join)

Example continued with concrete data:

```
trackedCustomers.collect().foreach(println)
// (101,((Ruetli,AG),Bern))
// (101,((Ruetli,AG),Thun))
// (102,((Brelaz,DemiTarif),Nyon))
// (102,((Brelaz,DemiTarif),Lausanne))
// (102,((Brelaz,DemiTarif),Geneve))
// (103,((Gress,DemiTarifVisa),St-Gallen))
// (103,((Gress,DemiTarifVisa),Chur))
// (103,((Gress,DemiTarifVisa),Zurich))
```

What happened to customer 104?

Inner Joins (join)

Example continued with concrete data:

```
trackedCustomers.collect().foreach(println)
// (101,((Ruetli,AG),Bern))
// (101,((Ruetli,AG),Thun))
// (102,((Brelaz,DemiTarif),Nyon))
// (102,((Brelaz,DemiTarif),Lausanne))
// (102,((Brelaz,DemiTarif),Geneve))
// (103,((Gress,DemiTarifVisa),St-Gallen))
// (103,((Gress,DemiTarifVisa),Chur))
// (103,((Gress,DemiTarifVisa),Zurich))
```

What happened to customer 104?

Customer 104 does *not* occur in the result, because there is no location data for this customer. Remember, inner joins require keys to occur in *both* source RDDs (i.e., we must have location info).

Outer Joins (`leftOuterJoin`, `rightOuterJoin`)

Outer joins return a new RDD containing combined pairs whose **keys don't have to be present in both input RDDs**.

Outer joins are particularly useful for customizing how the resulting joined RDD deals with missing keys. With outer joins, we can decide which RDD's keys are most essential to keep—the left, or the right RDD in the join expression.

```
def leftOuterJoin[W](other: RDD[(K, W)]): RDD[(K, (V, Option[W]))]
def rightOuterJoin[W](other: RDD[(K, W)]): RDD[(K, (Option[V], W))]
```

(Notice the insertion and position of the Option!)

Example: Let's assume the CFF wants to know for which subscribers the CFF has managed to collect location information. E.g., it's possible that someone has a demi-tarif, but doesn't use the CFF app and only pays cash for tickets.

Which join do we use?

Outer Joins (leftOuterJoin, rightOuterJoin)

Example continued with concrete data:

abos

```
(101, ("Ruetli", AG)),  
(102, ("Brelaz", DemiTarif)),  
(103, ("Gress", DemiTarifVisa)),  
(104, ("Schatten", DemiTarif))
```

locations

```
(101, "Bern"),  
(101, "Thun"),  
(102, "Lausanne"),  
(102, "Geneve"),  
(102, "Nyon"),  
(103, "Zurich"),  
(103, "St-Gallen"),  
(103, "Chur")
```

We want to combine both RDDs into one:

The CFF wants to know for which subscribers the CFF has managed to collect location information. E.g., it's possible that someone has a demi-tarif, but doesn't use the CFF app and only pays cash for tickets.

```
val abosWithOptionalLocations = ???
```

Outer Joins (`leftOuterJoin`, `rightOuterJoin`)

Example: Let's assume the CFF wants to know for which subscribers the CFF has managed to collect location information. E.g., it's possible that someone has a demi-tarif, but doesn't use the CFF app and only pays cash for tickets.

Which join do we use?

```
val abosWithOptionalLocations = ???
```

Outer Joins (leftOuterJoin, rightOuterJoin)

Example continued with concrete data:

abos

```
(101, ("Ruetli", AG)),  
(102, ("Brelaz", DemiTarif)),  
(103, ("Gress", DemiTarifVisa)),  
(104, ("Schatten", DemiTarif))
```

We want to make a new RDD with these!

locations

```
(101, "Bern"),  
(101, "Thun"),  
(102, "Lausanne"),  
(102, "Geneve"),  
(102, "Nyon"),  
(103, "Zurich"),  
(103, "St-Gallen"),  
(103, "Chur")
```

```
val abosWithOptionalLocations = ???
```

Outer Joins (`leftOuterJoin`, `rightOuterJoin`)

Example: Let's assume the CFF wants to know for which subscribers the CFF has managed to collect location information. E.g., it's possible that someone has a demi-tarif, but doesn't use the CFF app and only pays cash for tickets.

Which join do we use?

```
val abosWithOptionalLocations = abos.leftOuterJoin(locations)
// abosWithOptionalLocations: RDD[(Int, ((String, Abonnement), Option[String]))]
```

Outer Joins (leftOuterJoin, rightOuterJoin)

Example continued with concrete data:

```
abosWithOptionalLocations
(101,((Ruetli,AG),Some(Thun)))
(101,((Ruetli,AG),Some(Bern)))
(102,((Brelaz,DemiTarif),Some(Geneve)))
(102,((Brelaz,DemiTarif),Some(Nyon)))
(102,((Brelaz,DemiTarif),Some(Lausanne)))
(103,((Gress,DemiTarifVisa),Some(Zurich)))
(103,((Gress,DemiTarifVisa),Some(St-Gallen)))
(103,((Gress,DemiTarifVisa),Some(Chur)))
(104,((Schatten,DemiTarif),None))
```

customer# lastName kindOfAbo Option[frequentCity]

```
val abosWithOptionalLocations = abos.leftOuterJoin(locations)
// abosWithOptionalLocations: RDD[(Int, ((String, Abonnement), Option[String]))]
```

Outer Joins (leftOuterJoin, rightOuterJoin)

Example continued with concrete data:

```
val abosWithOptionalLocations = abos.leftOuterJoin(locations)
abosWithOptionalLocations.collect().foreach(println)
// (101,((Ruetli,AG),Some(Thun)))
// (101,((Ruetli,AG),Some(Bern)))
// (102,((Brelaz,DemiTarif),Some(Geneve)))
// (102,((Brelaz,DemiTarif),Some(Nyon)))
// (102,((Brelaz,DemiTarif),Some(Lausanne)))
// (103,((Gress,DemiTarifVisa),Some(Zurich)))
// (103,((Gress,DemiTarifVisa),Some(St-Gallen)))
// (103,((Gress,DemiTarifVisa),Some(Chur)))
// (104,((Schatten,DemiTarif),None))
```

Since we use a leftOuterJoin, keys are guaranteed to occur in the left source RDD. Therefore, in this case, we see customer 104 because that customer has a demi-tarif (the left RDD in the join).

Outer Joins (leftOuterJoin, rightOuterJoin)

We can do the converse using a rightOuterJoin.

abos

```
(101, ("Ruetli", AG)),  
(102, ("Brelaz", DemiTarif)),  
(103, ("Gress", DemiTarifVisa)),  
(104, ("Schatten", DemiTarif))
```

locations

```
(101, "Bern"),  
(101, "Thun"),  
(102, "Lausanne"),  
(102, "Geneve"),  
(102, "Nyon"),  
(103, "Zurich"),  
(103, "St-Gallen"),  
(103, "Chur")
```

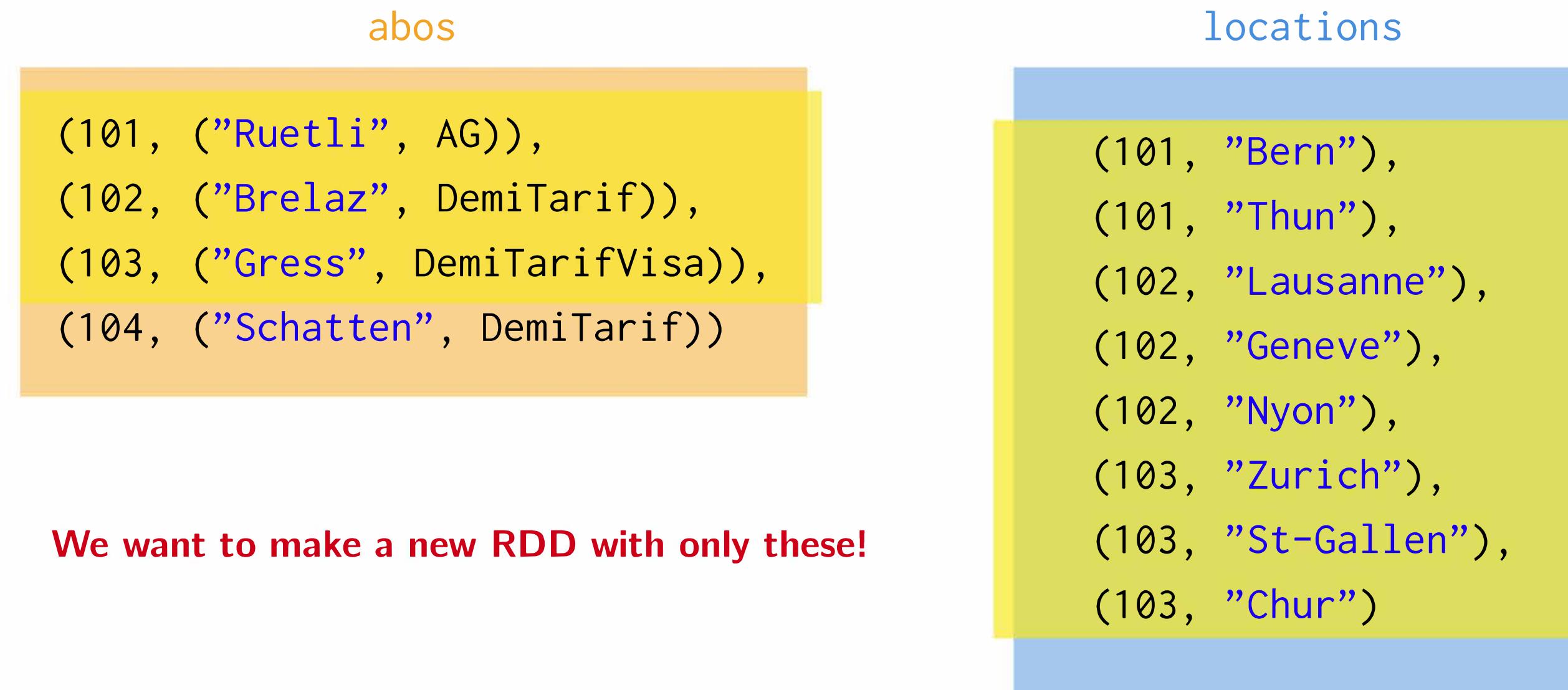
We want to combine both RDDs into one:

The CFF wants to know for which customers (smartphone app users) it has subscriptions for. E.g., it's possible that someone uses the mobile app, but has no demi-tarif.

```
val customersWithLocationDataAndOptionalAbos = ???
```

Outer Joins (leftOuterJoin, rightOuterJoin)

We can do the converse using a rightOuterJoin.



```
val customersWithLocationDataAndOptionalAbos = ???
```

Outer Joins (leftOuterJoin, rightOuterJoin)

We can do the converse using a rightOuterJoin.

Example: Let's assume in this case, the CFF wants to know for which customers (smartphone app users) it has subscriptions for. E.g., it's possible that someone uses the mobile app, but has no demi-tarif.

```
val customersWithLocationDataAndOptionalAbos =  
    abos.rightOuterJoin(locations)  
    // RDD[(Int, (Option[(String, Abonnement)], String))]
```

Outer Joins (leftOuterJoin, rightOuterJoin)

Example continued with concrete data:

customersWithLocationDataAndOptionalAbos

```
(101, (Some((Ruetli,AG)),Bern))  
(101, (Some((Ruetli,AG)),Thun))  
(102, (Some((Brelaz,DemiTarif)),Lausanne))  
(102, (Some((Brelaz,DemiTarif)),Geneve))  
(102, (Some((Brelaz,DemiTarif)),Nyon))  
(103, (Some((Gress,DemiTarifVisa)),Zurich))  
(103, (Some((Gress,DemiTarifVisa)),St-Gallen))  
(103, (Some((Gress,DemiTarifVisa)),Chur))
```

↑ ↑ ↑ ↑
customer# Option[(lastName,kindOfAbo)] frequentCity

```
val customersWithLocationDataAndOptionalAbos =  
    abos.rightOuterJoin(locations)  
    // RDD[(Int, (Option[(String, Abonnement)], String))]
```

Outer Joins (leftOuterJoin, rightOuterJoin)

Example continued with concrete data:

```
val customersWithLocationDataAndOptionalAbos =  
    abos.rightOuterJoin(locations)  
    // RDD[(Int, (Option[(String, Abonnement)], String))]  
  
customersWithLocationDataAndOptionalAbos.collect().foreach(println)  
// (101,(Some((Ruetli,AG)),Bern))  
// (101,(Some((Ruetli,AG)),Thun))  
// (102,(Some((Brelaz,DemiTarif)),Lausanne))  
// (102,(Some((Brelaz,DemiTarif)),Geneve))  
// (102,(Some((Brelaz,DemiTarif)),Nyon))  
// (103,(Some((Gress,DemiTarifVisa)),Zurich))  
// (103,(Some((Gress,DemiTarifVisa)),St-Gallen))  
// (103,(Some((Gress,DemiTarifVisa)),Chur))
```

Note that, here, customer 104 disappears again because that customer doesn't have location info stored with the CFF (the right RDD in the join).

?? org.apache.spark.rdd.RDD[(String, Int)] = ShuffledRDD[366] ??

Think again what happens when you have to do a groupBy or a groupByKey. Remember our data is distributed!

?? org.apache.spark.rdd.RDD[(String, Int)] = ShuffledRDD[366] ??

Think again what happens when you have to do a groupBy or a groupByKey. Remember our data is distributed!

We typically have to move data from one node to another to be “grouped with” its key. Doing this is called “shuffling”.

?? org.apache.spark.rdd.RDD[(String, Int)] = ShuffledRDD[366] ??

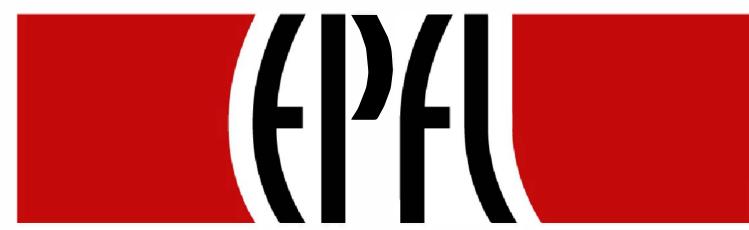
Think again what happens when you have to do a groupBy or a groupByKey. Remember our data is distributed!

We typically have to move data from one node to another to be “grouped with” its key. Doing this is called “shuffling”.

Shuffles Happen

Shuffles can be an enormous hit to because it means that Spark must send data from one node to another. Why? **Latency!**

We'll talk more about these in the next lecture.



ÉCOLE POLYTECHNIQUE
FÉDÉRALE DE LAUSANNE

Shuffling: What it is and why it's important

Big Data Analysis with Scala and Spark

Heather Miller

?? org.apache.spark.rdd.RDD[(String, Int)] = ShuffledRDD[366] ??

Think again what happens when you have to do a groupBy or a groupByKey.
Remember our data is distributed! **Did you notice anything odd?**

?? org.apache.spark.rdd.RDD[(String, Int)] = ShuffledRDD[366] ??

Think again what happens when you have to do a groupBy or a groupByKey.
Remember our data is distributed! **Did you notice anything odd?**

```
val pairs = sc.parallelize(List((1, "one"), (2, "two"), (3, "three")))
pairs.groupByKey()
// res2: org.apache.spark.rdd.RDD[Int, Iterable[String]]
//    = ShuffledRDD[16] at groupByKey at <console>:37
```

?? org.apache.spark.rdd.RDD[(String, Int)] = ShuffledRDD[366] ??

Think again what happens when you have to do a groupBy or a groupByKey.
Remember our data is distributed! **Did you notice anything odd?**

```
val pairs = sc.parallelize(List((1, "one"), (2, "two"), (3, "three")))
pairs.groupByKey()
// res2: org.apache.spark.rdd.RDD[Int, Iterable[String]]
//    = ShuffledRDD[16] at groupByKey at <console>:37
```

We typically have to move data from one node to another to be “grouped with” its key. Doing this is called “shuffling”.

?? org.apache.spark.rdd.RDD[(String, Int)] = ShuffledRDD[366] ??

Think again what happens when you have to do a groupBy or a groupByKey.
Remember our data is distributed! **Did you notice anything odd?**

```
val pairs = sc.parallelize(List((1, "one"), (2, "two"), (3, "three")))
pairs.groupByKey()
// res2: org.apache.spark.rdd.RDD[Int, Iterable[String]]
//    = ShuffledRDD[16] at groupByKey at <console>:37
```

We typically have to move data from one node to another to be “grouped with” its key. Doing this is called “shuffling”.

Shuffles Happen

Shuffles can be an enormous hit to because it means that Spark must send data from one node to another. Why? **Latency!**

Grouping and Reducing, Example

Let's start with an example. Given:

```
case class CFFPurchase(customerId: Int, destination: String, price: Double)
```

Assume we have an RDD of the purchases that users of the Swiss train company's, the CFF's, mobile app have made in the past month.

```
val purchasesRdd: RDD[CFFPurchase] = sc.textFile(...)
```

Goal: calculate how many trips, and how much money was spent by each individual customer over the course of the month.

Grouping and Reducing, Example

Goal: calculate how many trips, and how much money was spent by each individual customer over the course of the month.

```
val purchasesRdd: RDD[CFFPurchase] = sc.textFile(...)
```

```
val purchasesPerMonth = ...
```

Grouping and Reducing, Example

Goal: calculate how many trips, and how much money was spent by each individual customer over the course of the month.

```
val purchasesRdd: RDD[CFFPurchase] = sc.textFile(...)
```

```
val purchasesPerMonth =  
    purchasesRdd.map(p => (p.customerId, p.price)) // Pair RDD
```

Grouping and Reducing, Example

Goal: calculate how many trips, and how much money was spent by each individual customer over the course of the month.

```
val purchasesRdd: RDD[CFFPurchase] = sc.textFile(...)

val purchasesPerMonth =
  purchasesRdd.map(p => (p.customerId, p.price)) // Pair RDD
    .groupByKey() // groupByKey returns RDD[K, Iterable[V]]
```

Grouping and Reducing, Example

Goal: calculate how many trips, and how much money was spent by each individual customer over the course of the month.

```
val purchasesRdd: RDD[CFFPurchase] = sc.textFile(...)

// Returns: Array[(Int, (Int, Double))]

val purchasesPerMonth =
  purchasesRdd.map(p => (p.customerId, p.price)) // Pair RDD
    .groupByKey() // groupByKey returns RDD[(K, Iterable[V])]
    .map(p => (p._1, (p._2.size, p._2.sum)))
    .collect()
```

Grouping and Reducing, Example – What's Happening?

Let's start with an example dataset:

```
val purchases = List(CFFPurchase(100, "Geneva", 22.25),  
                     CFFPurchase(300, "Zurich", 42.10),  
                     CFFPurchase(100, "Fribourg", 12.40),  
                     CFFPurchase(200, "St. Gallen", 8.20),  
                     CFFPurchase(100, "Lucerne", 31.60),  
                     CFFPurchase(300, "Basel", 16.20))
```

What might the cluster look like with this data distributed over it?

Grouping and Reducing, Example – What's Happening?

What might the cluster look like with this data distributed over it?

Starting with purchasesRdd:

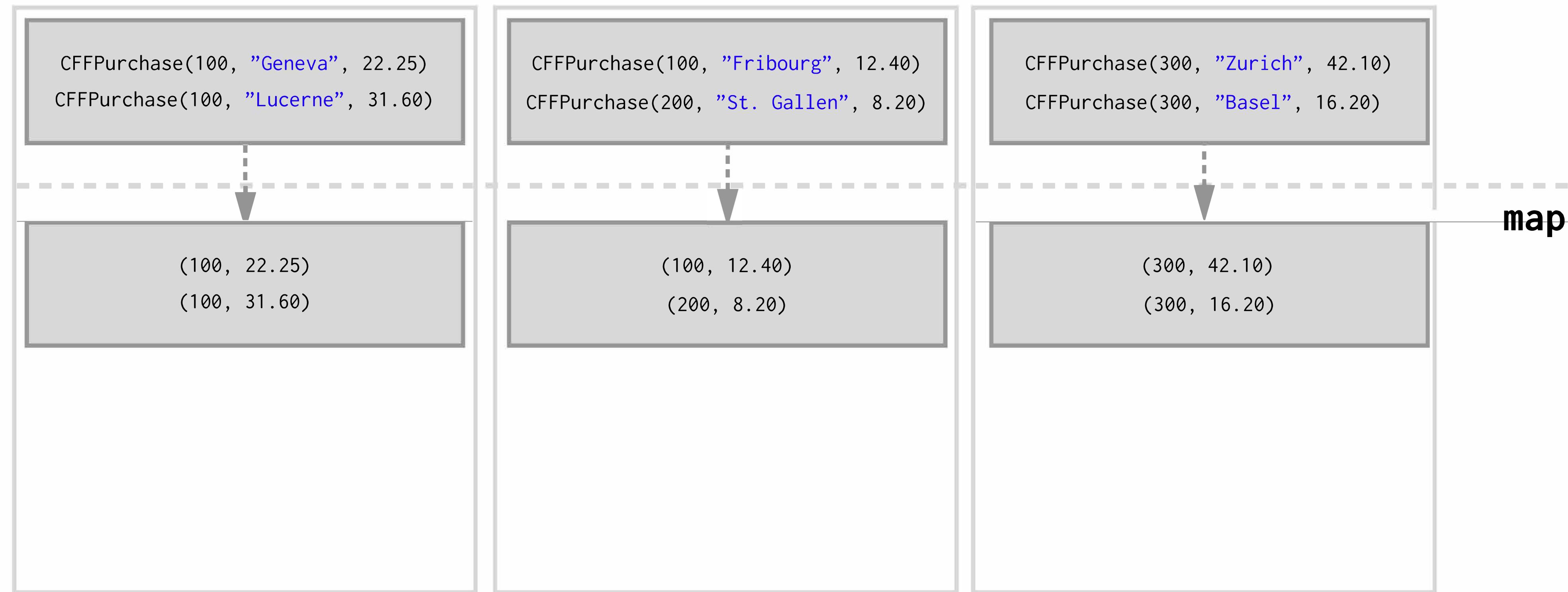
```
CFFPurchase(100, "Geneva", 22.25)  
CFFPurchase(100, "Lucerne", 31.60)
```

```
CFFPurchase(100, "Fribourg", 12.40)  
CFFPurchase(200, "St. Gallen", 8.20)
```

```
CFFPurchase(300, "Zurich", 42.10)  
CFFPurchase(300, "Basel", 16.20)
```

Grouping and Reducing, Example – What's Happening?

What might this look like on the cluster?



Grouping and Reducing, Example

Goal: calculate how many trips, and how much money was spent by each individual customer over the course of the month.

```
val purchasesRdd: RDD[CFFPurchase] = sc.textFile(...)

val purchasesPerMonth =
  purchasesRdd.map(p => (p.customerId, p.price)) // Pair RDD
    .groupByKey() // groupByKey returns RDD[K, Iterable[V]]
```

Grouping and Reducing, Example

Goal: calculate how many trips, and how much money was spent by each individual customer over the course of the month.

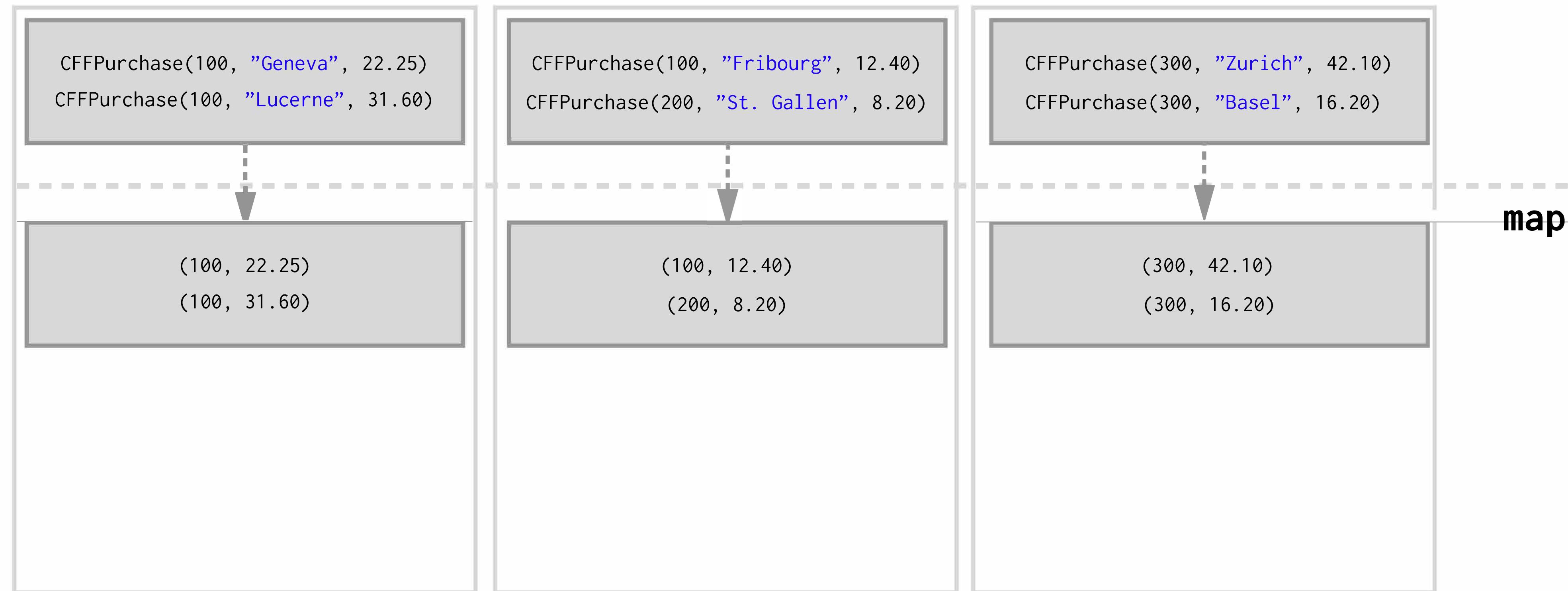
```
val purchasesRdd: RDD[CFFPurchase] = sc.textFile(...)

val purchasesPerMonth =
  purchasesRdd.map(p => (p.customerId, p.price)) // Pair RDD
    .groupByKey() // groupByKey returns RDD[K, Iterable[V]]
```

Note: groupByKey results in one key-value pair per key. And this single key-value pair cannot span across multiple worker nodes.

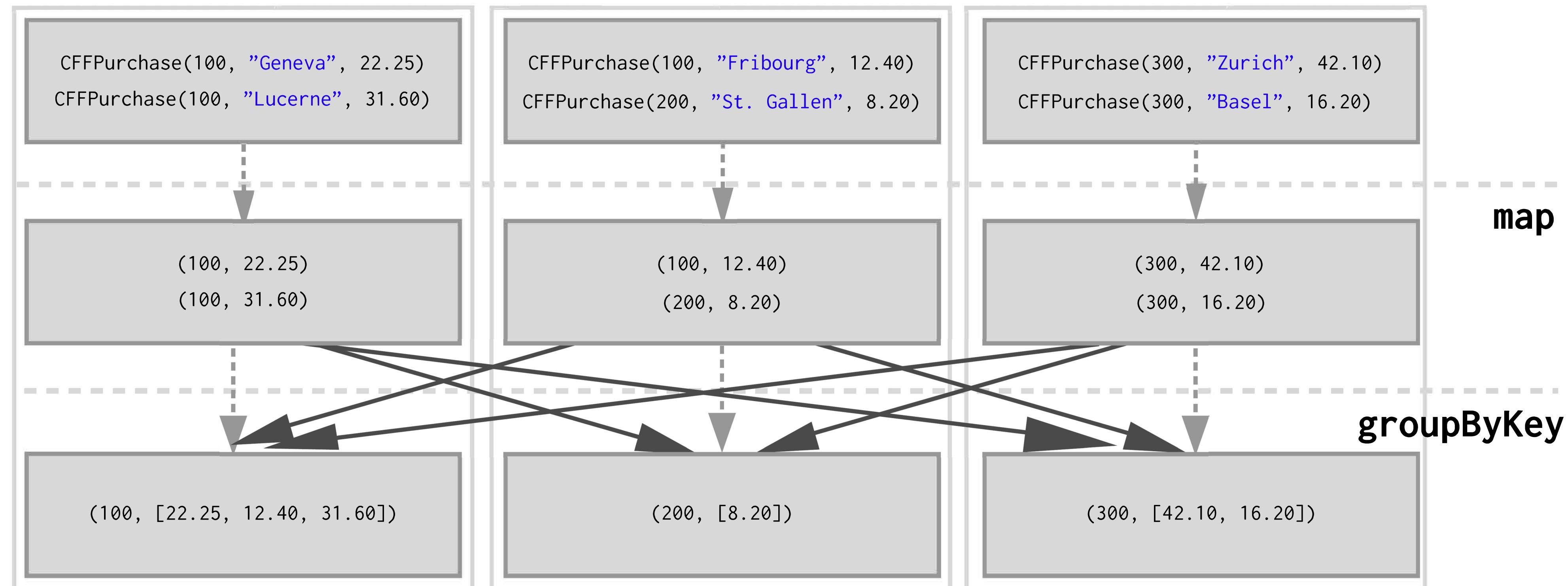
Grouping and Reducing, Example – What's Happening?

What might this look like on the cluster?



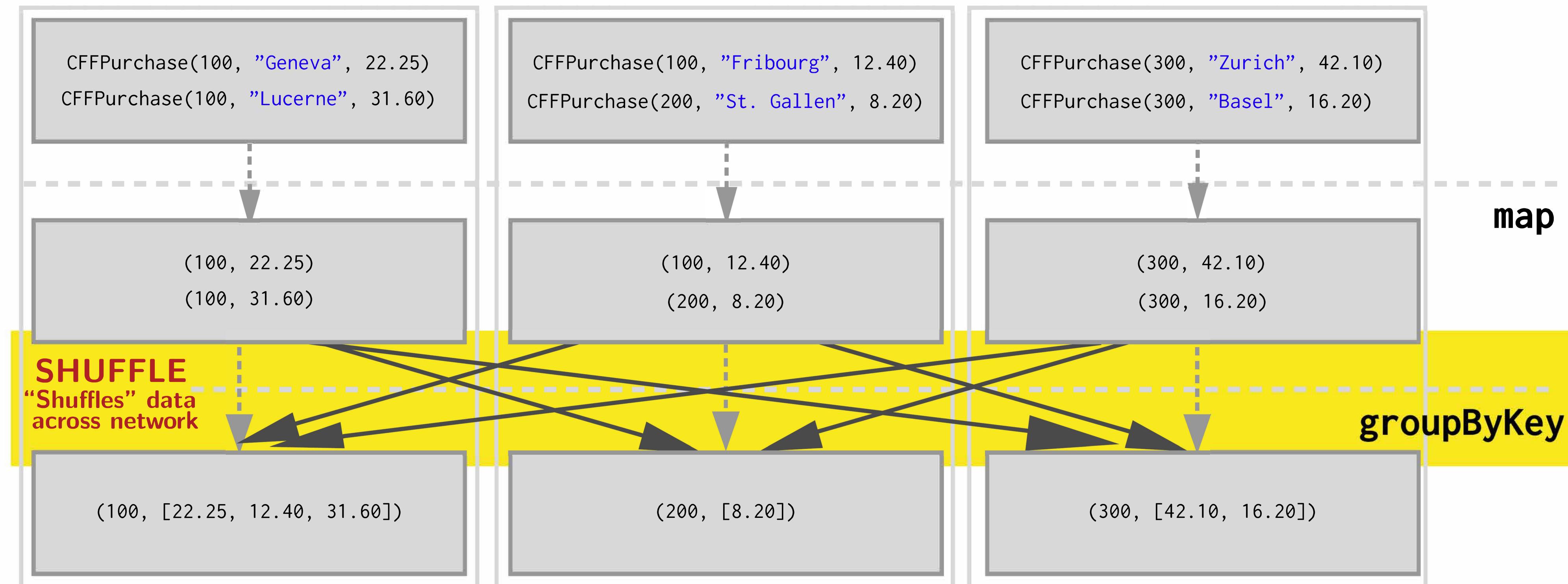
Grouping and Reducing, Example – What's Happening?

What might this look like on the cluster?



Grouping and Reducing, Example – What's Happening?

What might this look like on the cluster?



Reminder: Latency Matters (Humanized)

Shared Memory

Seconds

L1 cache reference.....0.5s
L2 cache reference.....7s
Mutex lock/unlock.....25s

Minutes

Main memory reference.....1m 40s

Distributed

Days

Roundtrip within
same datacenter.....5.8 days

Years

Send packet
CA->Netherlands->CA....4.8 years

We don't want to be sending all of our data over the network if it's not absolutely required. Too much network communication kills performance.

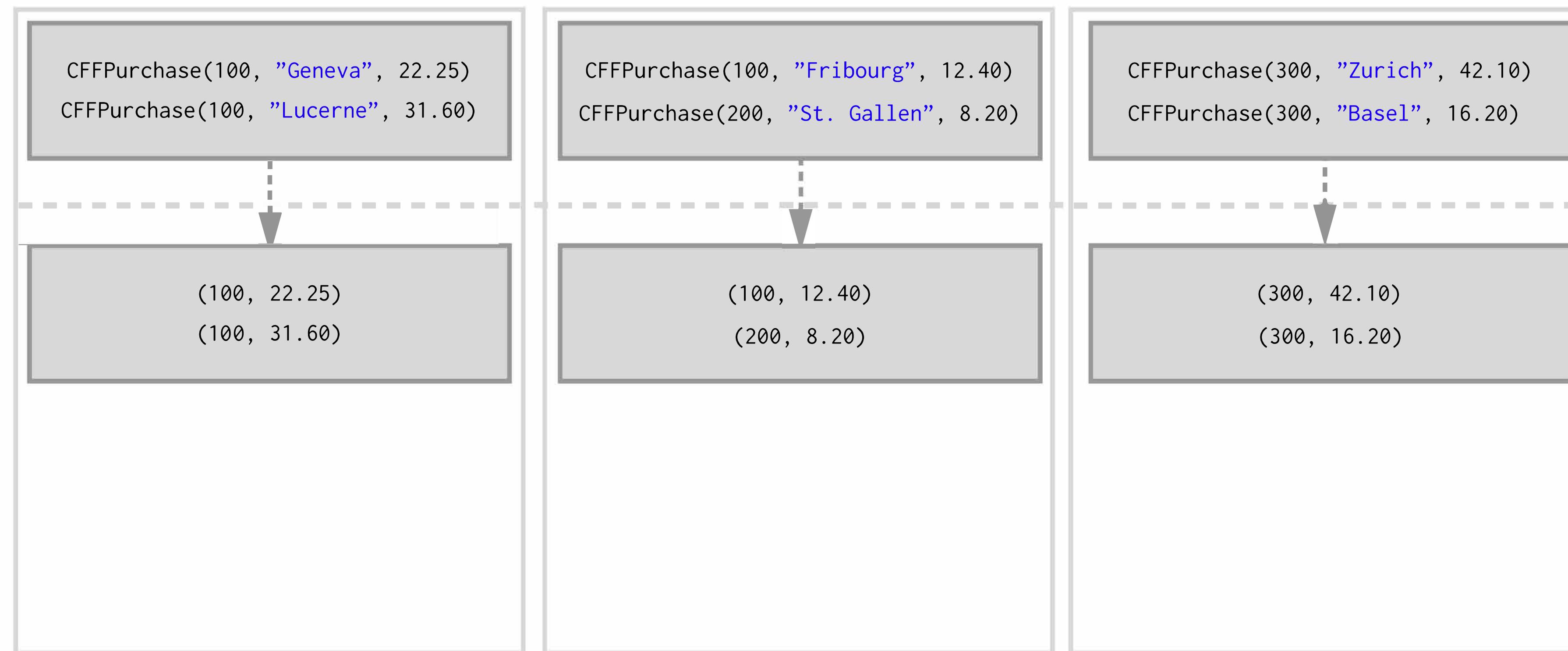
Can we do a better job?

Perhaps we don't need to send all pairs over the network.



Can we do a better job?

Perhaps we don't need to send all pairs over the network.



Perhaps we can reduce before we shuffle. This could greatly reduce the amount of data we have to send over the network.

Grouping and Reducing, Example – Optimized

We can use `reduceByKey`.

Conceptually, `reduceByKey` can be thought of as a combination of first doing `groupByKey` and then `reduce-ing` on all the values grouped per key. It's more efficient though, than using each separately. We'll see how in the following example.

Signature:

```
def reduceByKey(func: (V, V) => V): RDD[(K, V)]
```

Grouping and Reducing, Example – Optimized

Goal: calculate how many trips, and how much money was spent by each individual customer over the course of the month.

```
val purchasesRdd: RDD[CFFPurchase] = sc.textFile(...)

val purchasesPerMonth =
  purchasesRdd.map(p => (p.customerId, (1, p.price))) // Pair RDD
    .reduceByKey(...) // ?
```

Grouping and Reducing, Example – Optimized

Goal: calculate how many trips, and how much money was spent by each individual customer over the course of the month.

```
val purchasesRdd: RDD[CFFPurchase] = sc.textFile(...)

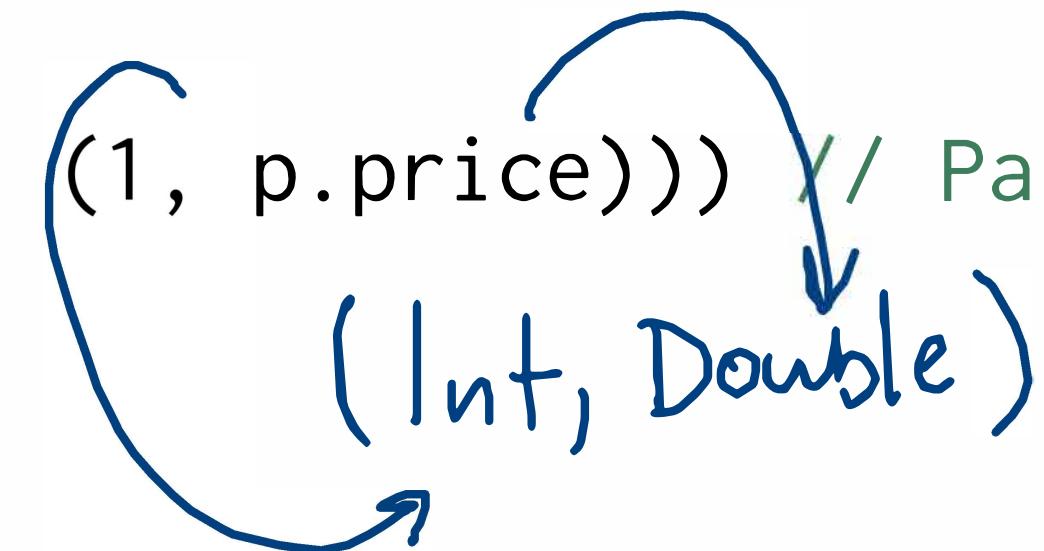
val purchasesPerMonth =
  purchasesRdd.map(p => (p.customerId, (1, p.price))) // Pair RDD
    .reduceByKey(...) // ?
```

Notice that the function passed to map has changed. It's now p => (p.customerId, (1, p.price)).

What function do we pass to reduceByKey in order to get a result that looks like: (customerId, (numTrips, totalSpent)) returned?

Grouping and Reducing, Example – Optimized

```
val purchasesPerMonth =  
    purchasesRdd.map(p => (p.customerId,  
        .reduceByKey(...) // ?
```



Grouping and Reducing, Example – Optimized

```
val purchasesPerMonth =  
    purchasesRdd.map(p => (p.customerId, (1, p.price))) // Pair RDD  
        .reduceByKey((v1, v2) => v1._1 + v2._1, v1._2 + v2._2)  
        .collect()  
    l + l  
    price + price
```

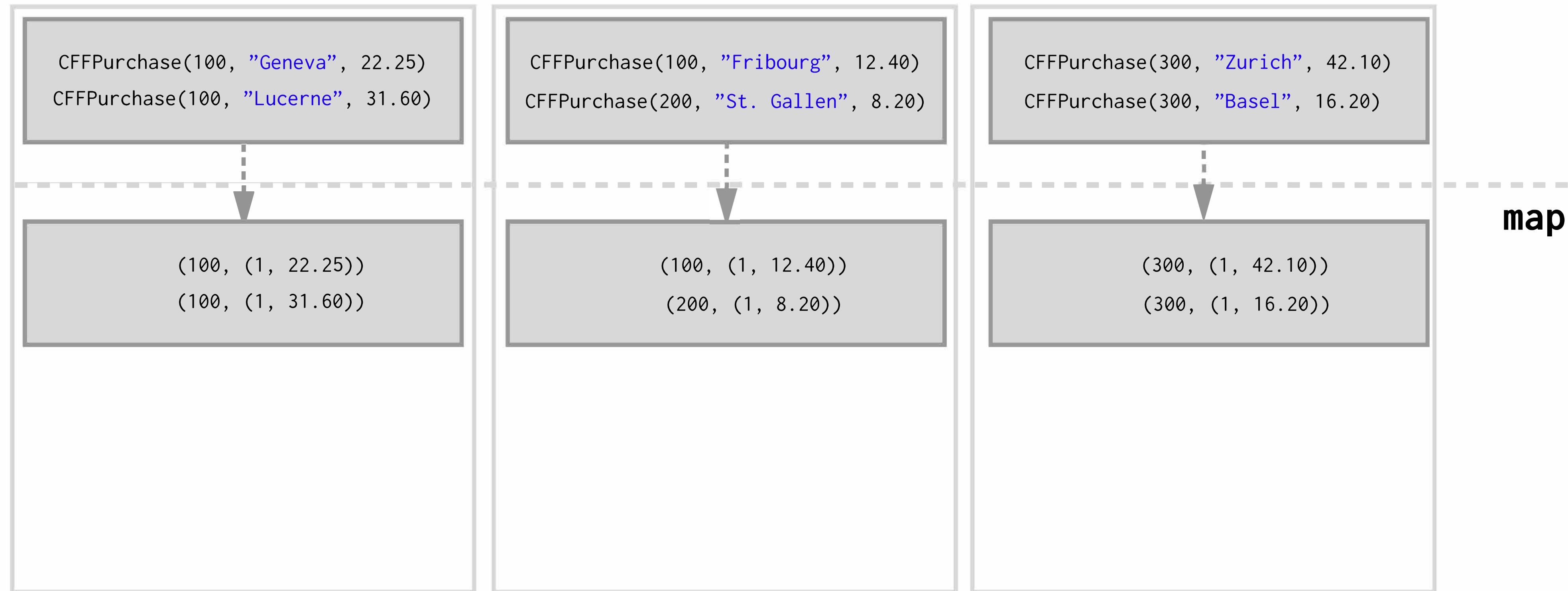
Grouping and Reducing, Example – Optimized

```
val purchasesPerMonth =  
    purchasesRdd.map(p => (p.customerId, (1, p.price))) // Pair RDD  
        .reduceByKey((v1, v2) => (v1._1 + v2._1, v1._2 + v2._2))  
        .collect()
```

What might this look like on the cluster?

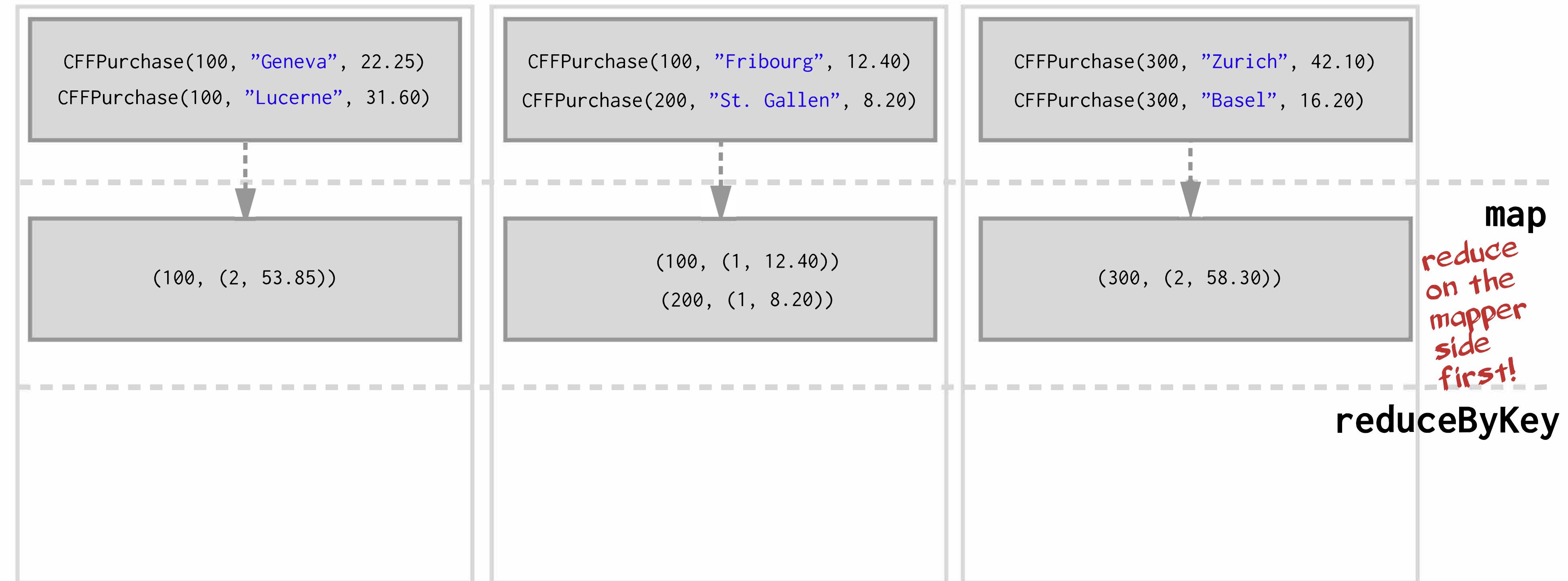
Grouping and Reducing, Example – Optimized

What might this look like on the cluster?



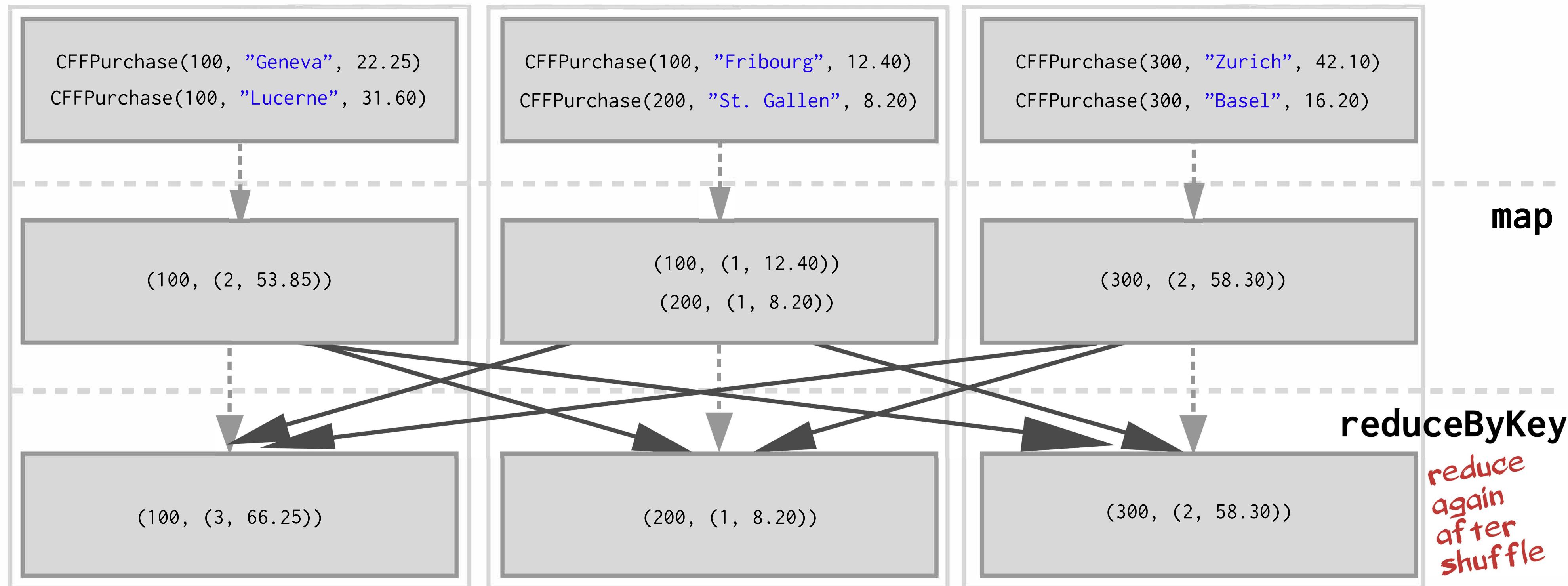
Grouping and Reducing, Example – Optimized

What might this look like on the cluster?



Grouping and Reducing, Example – Optimized

What might this look like on the cluster?



Grouping and Reducing, Example – Optimized

What are the benefits of this approach?

Grouping and Reducing, Example – Optimized

What are the benefits of this approach?

By reducing the dataset first, the amount of data sent over the network during the shuffle is greatly reduced.

This can result in non-trivial gains in performance!

Grouping and Reducing, Example – Optimized

What are the benefits of this approach?

By reducing the dataset first, the amount of data sent over the network during the shuffle is greatly reduced.

This can result in non-trivial gains in performance!

Let's benchmark on a real cluster.

groupByKey and reduceByKey Running Times

```
> val purchasesPerMonthSlowLarge = purchasesRddLarge.map(p => (p.customerId, p.price))
   .groupByKey()
   .map(p => (p._1, (p._2.size, p._2.sum)))
   .count()
```

purchasesPerMonthSlowLarge: Long = 100000

Command took 15.48s

```
> val purchasesPerMonthFastLarge = purchasesRddLarge.map(p => (p.customerId, (1, p.price)))
   .reduceByKey((v1, v2) => (v1._1 + v2._1, v1._2 + v2._2))
   .count()
```

purchasesPerMonthFastLarge: Long = 100000

Command took 4.65s

Shuffling

Recall our example using groupByKey:

```
val purchasesPerCust =  
    purchasesRdd.map(p => (p.customerId, p.price)) // Pair RDD  
        .groupByKey()
```

Shuffling

Recall our example using groupByKey:

```
val purchasesPerCust =  
    purchasesRdd.map(p => (p.customerId, p.price)) // Pair RDD  
        .groupByKey()
```

Grouping all values of key-value pairs with the same key requires collecting all key-value pairs with the same key on the same machine.

But how does Spark know which key to put on which machine?

Shuffling

Recall our example using `groupByKey`:

```
val purchasesPerCust =  
    purchasesRdd.map(p => (p.customerId, p.price)) // Pair RDD  
        .groupByKey()
```

Grouping all values of key-value pairs with the same key requires collecting all key-value pairs with the same key on the same machine.

But how does Spark know which key to put on which machine?

- ▶ By default, Spark uses *hash partitioning* to determine which key-value pair should be sent to which machine.



ÉCOLE POLYTECHNIQUE
FÉDÉRALE DE LAUSANNE

Partitioning

Big Data Analysis with Scala and Spark

Heather Miller

“Partitioning”?

In the last session, we were looking at an example involving `groupByKey`, before we discovered that this operation causes data to be *shuffled* over the network.

Grouping all values of key-value pairs with the same key requires collecting all key-value pairs with the same key on the same machine.

We concluded the last session asking ourselves,

But how does Spark know which key to put on which machine?

Before we try to optimize that example any further, let's first take a quick detour into what partitioning is...

Partitions

The data within an RDD is split into several *partitions*.

Properties of partitions:

- ▶ Partitions never span multiple machines, i.e., tuples in the same partition are guaranteed to be on the same machine.
- ▶ Each machine in the cluster contains one or more partitions.
- ▶ The number of partitions to use is configurable. By default, it equals the *total number of cores on all executor nodes*.

Two kinds of partitioning available in Spark:

- ▶ Hash partitioning
- ▶ Range partitioning

Customizing a partitioning is only possible on Pair RDDs.

Hash partitioning

Back to our example. Given a Pair RDD that should be grouped:

```
val purchasesPerCust =  
    purchasesRdd.map(p => (p.customerId, p.price)) // Pair RDD  
        .groupByKey()
```

Hash partitioning

Back to our example. Given a Pair RDD that should be grouped:

```
val purchasesPerCust =  
    purchasesRdd.map(p => (p.customerId, p.price)) // Pair RDD  
        .groupByKey()
```

groupByKey first computes per tuple (k, v) its partition p :

```
p = k.hashCode() % numPartitions
```

Then, all tuples in the same partition p are sent to the machine hosting p .

Intuition: hash partitioning attempts to spread data evenly across partitions *based on the key*.

Range partitioning

Pair RDDs may contain keys that have an *ordering* defined.

- ▶ Examples: Int, Char, String, ...

For such RDDs, *range partitioning* may be more efficient.

Using a range partitioner, keys are partitioned according to:

1. an *ordering* for keys
2. a set of *sorted ranges* of keys

Property: tuples with keys in the same range appear on the same machine.

Hash Partitioning: Example

Consider a Pair RDD, with keys [8, 96, 240, 400, 401, 800], and a desired number of partitions of 4.

Hash Partitioning: Example

Consider a Pair RDD, with keys [8, 96, 240, 400, 401, 800], and a desired number of partitions of 4.

Furthermore, suppose that hashCode() is the identity ($n.hashCode() == n$).

$$\begin{aligned} p &= K \bmod numPartitions \\ &= K \% 4 \end{aligned}$$

Hash Partitioning: Example

Consider a Pair RDD, with keys [8, 96, 240, 400, 401, 800], and a desired number of partitions of 4.

Furthermore, suppose that hashCode() is the identity (`n.hashCode() == n`).

In this case, hash partitioning distributes the keys as follows among the partitions:

- ▶ partition 0: [8, 96, 240, 400, 800]
- ▶ partition 1: [401]
- ▶ partition 2: []
- ▶ partition 3: []

$$p = K \% \Psi$$

The result is a very unbalanced distribution which hurts performance.

Range Partitioning: Example

Using *range partitioning* the distribution can be improved significantly:

- ▶ Assumptions: (a) keys non-negative, (b) 800 is biggest key in the RDD.
- ▶ Set of ranges: [1, 200], [201, 400], [401, 600], [601, 800]

Range Partitioning: Example

Using *range partitioning* the distribution can be improved significantly:

- ▶ Assumptions: (a) keys non-negative, (b) 800 is biggest key in the RDD.
- ▶ Set of ranges: [1, 200], [201, 400], [401, 600], [601, 800]

In this case, range partitioning distributes the keys as follows among the partitions:

- ▶ partition 0: [8, 96]
- ▶ partition 1: [240, 400]
- ▶ partition 2: [401]
- ▶ partition 3: [800]

The resulting partitioning is much more balanced.

Partitioning Data

How do we set a partitioning for our data?

Partitioning Data

How do we set a partitioning for our data?

There are two ways to create RDDs with specific partitionings:

1. Call `partitionBy` on an RDD, providing an explicit Partitioner.
2. Using transformations that return RDDs with specific partitioners.

Partitioning Data: `partitionBy`

Invoking `partitionBy` creates an RDD with a specified partitioner.

Partitioning Data: `partitionBy`

Invoking `partitionBy` creates an RDD with a specified partitioner.

Example:

```
val pairs = purchasesRdd.map(p => (p.customerId, p.price))
```

Partitioning Data: `partitionBy`

Invoking `partitionBy` creates an RDD with a specified partitioner.

Example:

```
val pairs = purchasesRdd.map(p => (p.customerId, p.price))
```

```
val tunedPartitioner = new RangePartitioner(8, pairs)
```

```
val partitioned = pairs.partitionBy(tunedPartitioner).persist()
```



Partitioning Data: `partitionBy`

Invoking `partitionBy` creates an RDD with a specified partitioner.

Example:

```
val pairs = purchasesRdd.map(p => (p.customerId, p.price))
```

```
val tunedPartitioner = new RangePartitioner(8, pairs)
```

```
val partitioned = pairs.partitionBy(tunedPartitioner).persist()
```

Creating a `RangePartitioner` requires:

1. Specifying the desired number of partitions.
2. Providing a Pair RDD with *ordered keys*. This RDD is *sampled* to create a suitable set of *sorted ranges*.

Partitioning Data: `partitionBy`

Invoking `partitionBy` creates an RDD with a specified partitioner.

Example:

```
val pairs = purchasesRdd.map(p => (p.customerId, p.price))
```

```
val tunedPartitioner = new RangePartitioner(8, pairs)
```

```
val partitioned = pairs.partitionBy(tunedPartitioner).persist()
```

Creating a RangePartitioner requires:

1. Specifying the desired number of partitions.
2. Providing a Pair RDD with *ordered keys*. This RDD is *sampled* to create a suitable set of *sorted ranges*.

Important: the result of `partitionBy` should be persisted. Otherwise, the partitioning is repeatedly applied (involving shuffling!) each time the partitioned RDD is used.

Partitioning Data Using Transformations

Partitioner from parent RDD:

Pair RDDs that are the result of a transformation on a *partitioned* Pair RDD typically is configured to use the hash partitioner that was used to construct it.

Automatically-set partitioners:

Some operations on RDDs automatically result in an RDD with a known partitioner – for when it makes sense.

For example, by default, when using `sortByKey`, a `RangePartitioner` is used. Further, the default partitioner when using `groupByKey`, is a `HashPartitioner`, as we saw earlier.

Partitioning Data Using Transformations

Operations on Pair RDDs that hold to (and propagate) a partitioner:

- ▶ cogroup
- ▶ groupWith
- ▶ join
- ▶ leftOuterJoin
- ▶ rightOuterJoin
- ▶ groupByKey
- ▶ reduceByKey
- ▶ foldByKey
- ▶ combineByKey
- ▶ partitionBy
- ▶ sort
- ▶ mapValues (if parent has a partitioner)
- ▶ flatMapValues (if parent has a partitioner)
- ▶ filter (if parent has a partitioner)

All other operations will produce a result without a partitioner.

Partitioning Data Using Transformations

...All other operations will produce a result without a partitioner.

Why?

Partitioning Data Using Transformations

...All other operations will produce a result without a partitioner.

Why?

Consider the map transformation. Given that we have a hash partitioned Pair RDD, why would it make sense for map to lose the partitioner in its result RDD?

Partitioning Data Using Transformations

...All other operations will produce a result without a partitioner.

Why?

Consider the map transformation. Given that we have a hash partitioned Pair RDD, why would it make sense for map to lose the partitioner in its result RDD?

Because it's possible for map to change the key . *E.g.,:*



Partitioning Data Using Transformations

...All other operations will produce a result without a partitioner.

Why?

Consider the map transformation. Given that we have a hash partitioned Pair RDD, why would it make sense for map to lose the partitioner in its result RDD?

Because it's possible for map to change the key . *E.g.,:*

```
rdd.map((k: String, v: Int) => ("doh!", v))
```

In this case, if the map transformation preserved the partitioner in the result RDD, it no longer make sense, as now the keys are all different.

Hence mapValues. It enables us to still do map transformations without changing the keys, thereby preserving the partitioner.



ÉCOLE POLYTECHNIQUE
FÉDÉRALE DE LAUSANNE

Optimizing with Partitioners

Big Data Analysis with Scala and Spark

Heather Miller

Optimizing with Partitioners

We saw in the last session that Spark makes a few kinds of partitioners available out-of-the-box to users:

- ▶ **hash partitioners** and
- ▶ **range partitioners**.

We also learned what kinds of operations may introduce new partitioners, or which may discard custom partitioners.

However, we haven't covered *why* someone would want to repartition their data.

Optimizing with Partitioners

We saw in the last session that Spark makes a few kinds of partitioners available out-of-the-box to users:

- ▶ **hash partitioners** and
- ▶ **range partitioners**.

We also learned what kinds of operations may introduce new partitioners, or which may discard custom partitioners.

However, we haven't covered *why* someone would want to repartition their data.

Partitioning can bring substantial performance gains, especially in the face of shuffles.

Optimization using range partitioning

Using range partitioners we can optimize our earlier use of reduceByKey so that it does not involve any shuffling over the network at all!

Optimization using range partitioning

Using range partitioners we can optimize our earlier use of reduceByKey so that it does not involve any shuffling over the network at all!

```
val pairs = purchasesRdd.map(p => (p.customerId, p.price))
```

```
val tunedPartitioner = new RangePartitioner(8, pairs)
```

```
val partitioned = pairs.partitionBy(tunedPartitioner)  
    .persist()
```

```
val purchasesPerCust =  
    partitioned.map(p => (p._1, (1, p._2)))
```

```
val purchasesPerMonth = purchasesPerCust  
    .reduceByKey((v1, v2) => (v1._1 + v2._1, v1._2 + v2._2))  
    .collect()
```

Optimization using range partitioning

```
> val purchasesPerMonthSlowLarge = purchasesRddLarge.map(p => (p.customerId, p.price))
   .groupByKey()
   .map(p => (p._1, (p._2.size, p._2.sum)))
   .count()
```

purchasesPerMonthSlowLarge: Long = 100000

Command took 15.48s

```
> val purchasesPerMonthFastLarge = purchasesRddLarge.map(p => (p.customerId, (1, p.price)))
   .reduceByKey((v1, v2) => (v1._1 + v2._1, v1._2 + v2._2))
   .count()
```

purchasesPerMonthFastLarge: Long = 100000

Command took 4.65s

On the range partitioned data:

```
> val purchasesPerMonthFasterLarge = partitioned.map(x => x)
   .reduceByKey((v1, v2) => (v1._1 + v2._1, v1._2 + v2._2))
   .count()
```

purchasesPerMonthFasterLarge: Long = 100000

Command took 1.79s

Optimization using range partitioning

```
> val purchasesPerMonthSlowLarge = purchasesRddLarge.map(p => (p.customerId, p.price))
   .groupByKey()
   .map(p => (p._1, (p._2.size, p._2.sum)))
   .count()
```

purchasesPerMonthSlowLarge: Long = 100000

Command took 15.48s

```
> val purchasesPerMonthFastLarge = purchasesRddLarge.map(p => (p.customerId, (1, p.price)))
   .reduceByKey((v1, v2) => (v1._1 + v2._1, v1._2 + v2._2))
   .count()
```

purchasesPerMonthFastLarge: Long = 100000

Command took 4.65s

On the range partitioned data:

```
> val purchasesPerMonthFasterLarge = partitioned.map(x => x)
   .reduceByKey((v1, v2) => (v1._1 + v2._1, v1._2 + v2._2))
   .count()
```

purchasesPerMonthFasterLarge: Long = 100000

Command took 1.79s almost a 9x speedup over
purchasePerMonthSlowLarge!

Partitioning Data: `partitionBy`, Another Example

From pages 61-64 of the Learning Spark book

Consider an application that keeps a large table of user information in memory:

- ▶ `userData` - **BIG**, containing (`UserID`, `UserInfo`) pairs, where `UserInfo` contains a list of topics the user is subscribed to.

The application periodically combines this **big** table with a smaller file representing events that happened in the past five minutes.

- ▶ `events` – *small*, containing (`UserID`, `LinkInfo`) pairs for users who have clicked a link on a website in those five minutes:

For example, we may wish to count how many users visited a link that was not to one of their subscribed topics. We can perform this combination with Spark's `join` operation, which can be used to group the `UserInfo` and `LinkInfo` pairs for each `UserID` by key.

Partitioning Data: `partitionBy`, Another Example

From pages 61-64 of the Learning Spark book

```
val sc = new SparkContext(...)  
val userData = sc.sequenceFile[UserID, UserInfo]("hdfs://...").persist()  
  
def processNewLogs(logFileName: String) {  
    val events = sc.sequenceFile[UserID, LinkInfo](logFileName)  
    val joined = userData.join(events) //RDD of (UserID, (UserInfo, LinkInfo))  
    val offTopicVisits = joined.filter {  
        case (userId, (userInfo, linkInfo)) => // Expand the tuple  
            !userInfo.topics.contains(linkInfo.topic)  
    }.count()  
    println("Number of visits to non-subscribed topics: " + offTopicVisits)  
}
```

Is this OK?

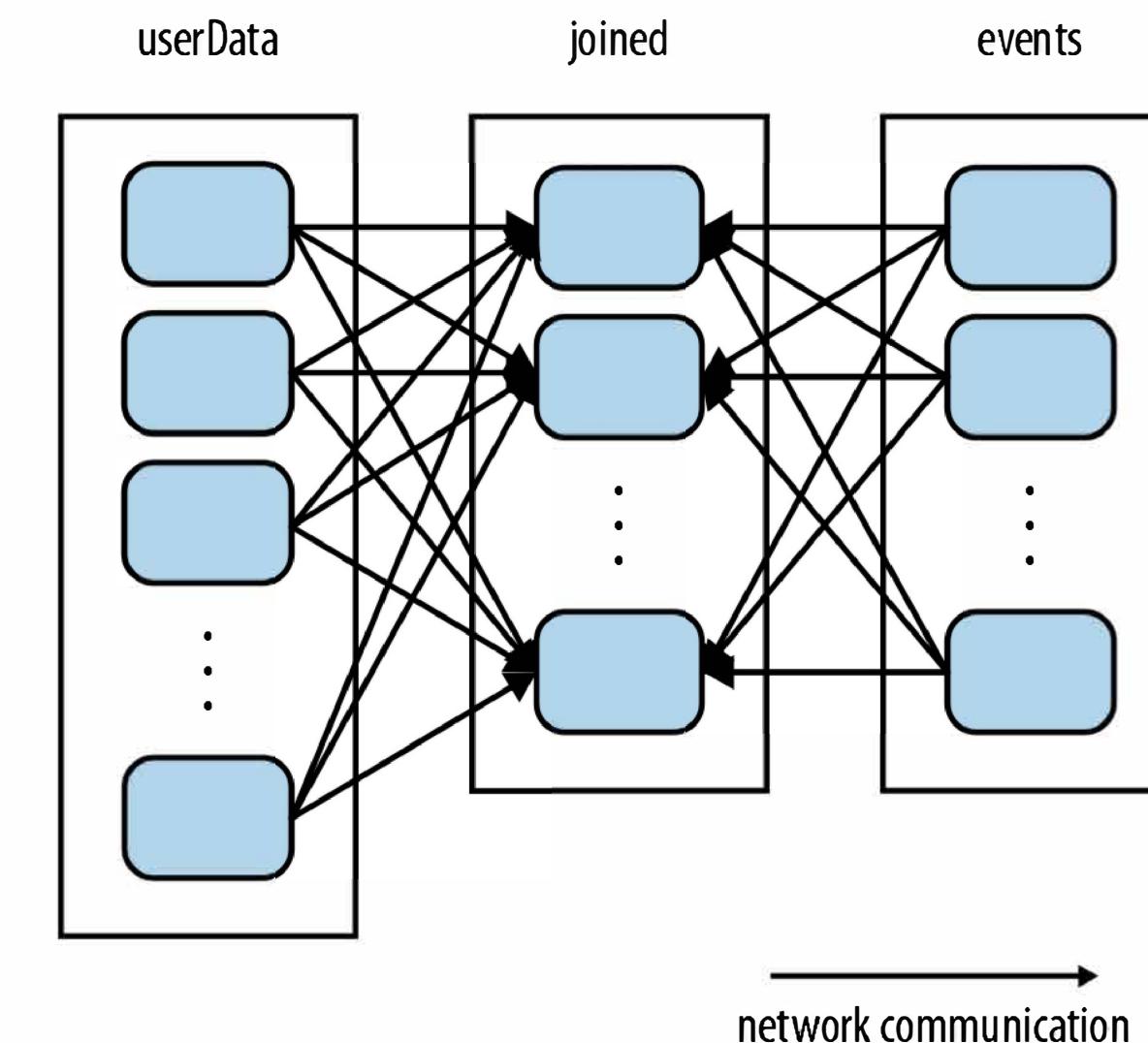
Partitioning Data: `partitionBy`, Another Example

From pages 61-64 of the Learning Spark book

It will be very inefficient!

Why? The join operation, called each time `processNewLogs` is invoked, does not know anything about how the keys are partitioned in the datasets.

By default, this operation will hash all the keys of both datasets, sending elements with the same key hash across the network to the same machine, and then join together the elements with the same key on that machine. **Even though `userData` doesn't change!**



Partitioning Data: `partitionBy`, Another Example

Fixing this is easy. Just use `partitionBy` on the **big** `userData` RDD at the start of the program!

Partitioning Data: `partitionBy`, Another Example

Fixing this is easy. Just use `partitionBy` on the **big** `userData` RDD at the start of the program!

Therefore, `userData` becomes:

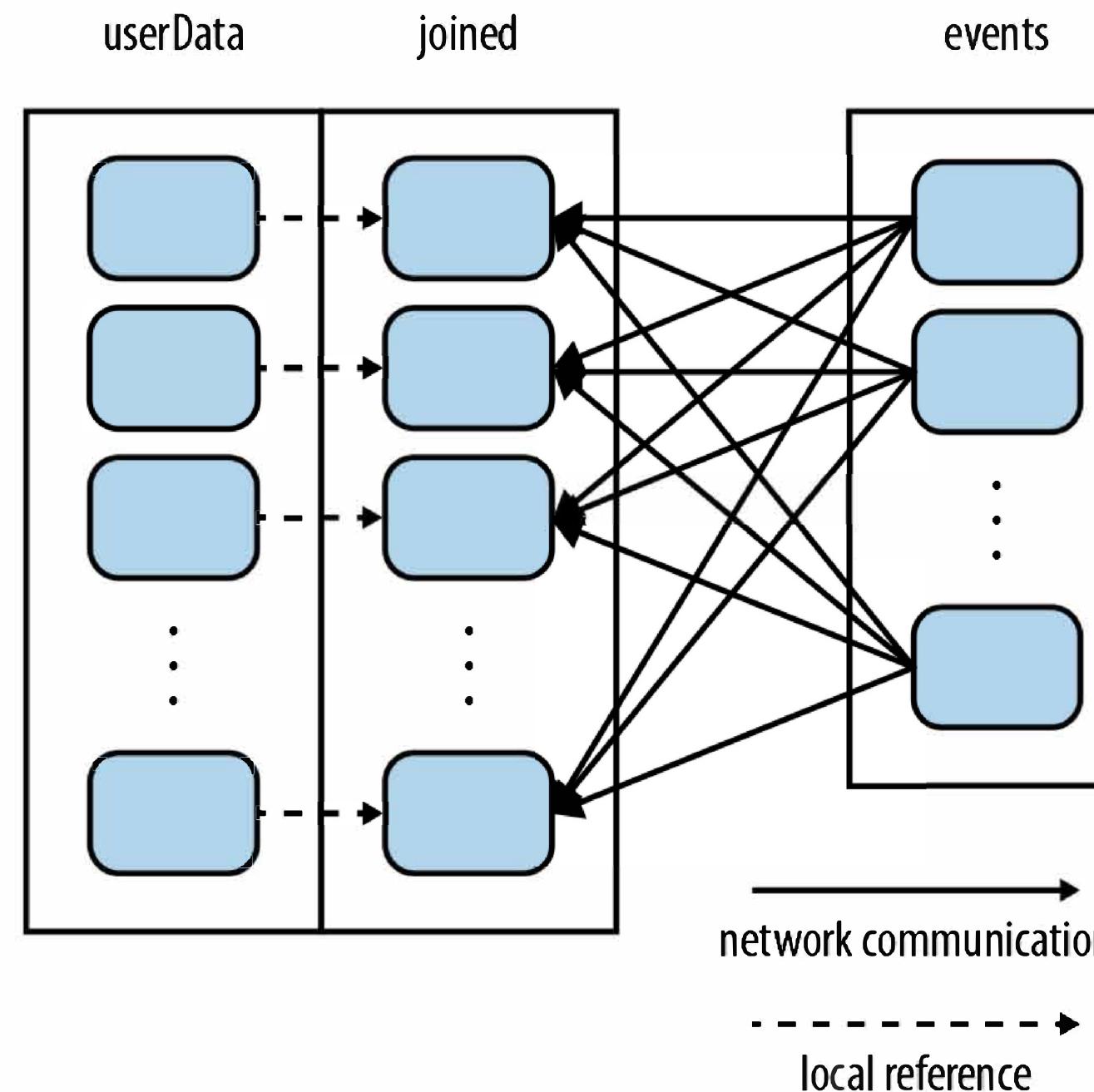
```
val userData = sc.sequenceFile[UserID, UserInfo]("hdfs://...")  
    .partitionBy(new HashPartitioner(100)) // Create 100 partitions  
    .persist()
```

Since we called `partitionBy` when building `userData`, Spark will now know that it is hash-partitioned, and calls to join on it will take advantage of this information.

In particular, when we call `userData.join(events)`, Spark will shuffle only the events RDD, sending events with each particular `UserID` to the machine that contains the corresponding hash partition of `userData`.

Partitioning Data: `partitionBy`, Another Example

Or, shown visually:



Now that `userData` is pre-partitioned, Spark will shuffle only the `events` RDD, sending events with each particular UserID to the machine that contains the corresponding hash partition of `userData`.

Back to shuffling

Recall our example using groupByKey:

```
val purchasesPerCust =  
    purchasesRdd.map(p => (p.customerId, p.price)) // Pair RDD  
        .groupByKey()
```

Back to shuffling

Recall our example using groupByKey:

```
val purchasesPerCust =  
    purchasesRdd.map(p => (p.customerId, p.price)) // Pair RDD  
        .groupByKey()
```

Grouping all values of key-value pairs with the same key requires collecting all key-value pairs with the same key on the same machine.

Back to shuffling

Recall our example using `groupByKey`:

```
val purchasesPerCust =  
    purchasesRdd.map(p => (p.customerId, p.price)) // Pair RDD  
        .groupByKey()
```

Grouping all values of key-value pairs with the same key requires collecting all key-value pairs with the same key on the same machine.

Grouping is done using a hash partitioner with default parameters.

Back to shuffling

Recall our example using `groupByKey`:

```
val purchasesPerCust =  
    purchasesRdd.map(p => (p.customerId, p.price)) // Pair RDD  
        .groupByKey()
```

Grouping all values of key-value pairs with the same key requires collecting all key-value pairs with the same key on the same machine.

Grouping is done using a hash partitioner with default parameters.

The result RDD, `purchasesPerCust`, is configured to use the hash partitioner that was used to construct it.

How do I know a shuffle will occur?

Rule of thumb: a shuffle *can* occur when the resulting RDD depends on other elements from the same RDD or another RDD.

How do I know a shuffle will occur?

Rule of thumb: a shuffle *can* occur when the resulting RDD depends on other elements from the same RDD or another RDD.

Note: sometimes one can be clever and avoid much or all network communication while still using an operation like join via smart partitioning

How do I know a shuffle will occur?

You can also figure out whether a shuffle has been planned/executed via:

1. The return type of certain transformations, e.g.,

```
org.apache.spark.rdd.RDD[(String, Int)] = ShuffledRDD[366]
```

2. Using function `toDebugString` to see its execution plan:

```
partitioned.reduceByKey((v1, v2) => (v1._1 + v2._1, v1._2 + v2._2))
  .toDebugString
res9: String =
(8) MapPartitionsRDD[622] at reduceByKey at <console>:49 []
  |  ShuffledRDD[615] at partitionBy at <console>:48 []
  |    CachedPartitions: 8; MemorySize: 1754.8 MB; DiskSize: 0.0 B
```

Operations that *might* cause a shuffle

- ▶ cogroup
- ▶ groupWith
- ▶ join
- ▶ leftOuterJoin
- ▶ rightOuterJoin
- ▶ groupByKey
- ▶ reduceByKey
- ▶ combineByKey
- ▶ distinct
- ▶ intersection
- ▶ repartition
- ▶ coalesce

Avoiding a Network Shuffle By Partitioning

There are a few ways to use operations that *might* cause a shuffle and to still avoid much or all network shuffling.

Can you think of an example?

Avoiding a Network Shuffle By Partitioning

There are a few ways to use operations that *might* cause a shuffle and to still avoid much or all network shuffling.

Can you think of an example?

2 Examples:

1. `reduceByKey` running on a pre-partitioned RDD will cause the values to be computed *locally*, requiring only the final reduced value has to be sent from the worker to the driver.
2. `join` called on two RDDs that are pre-partitioned with the same partitioner and cached on the same machine will cause the join to be computed *locally*, with no shuffling across the network.

Shuffles Happen: Key Takeaways

How your data is organized on the cluster, and what operations you're doing with it matters!

We've seen speedups of 10x on small examples just by trying to ensure that data is not transmitted over the network to other machines.

This can hugely affect your day job if you're trying to run a job that should run in 4 hours, but due to a missed opportunity to partition data or optimize away a shuffle, it could take **40 hours** instead.



ÉCOLE POLYTECHNIQUE
FÉDÉRALE DE LAUSANNE

Wide vs Narrow Dependencies

Big Data Analysis with Scala and Spark

Heather Miller

Not All Transformations are Created Equal

Some transformations significantly more expensive (latency) than others

E.g., requiring lots of data to be transferred over the network, sometimes unnecessarily.

Not All Transformations are Created Equal

Some transformations significantly more expensive (latency) than others

E.g., requiring lots of data to be transferred over the network, sometimes unnecessarily.

In the past sessions:

- ▶ we learned that shuffling sometimes happens on some transformations.

In this session:

- ▶ we'll look at how RDDs are represented.
- ▶ we'll dive into how and when Spark decides it must shuffle data.
- ▶ we'll see how these dependencies make fault tolerance possible.

Lineages

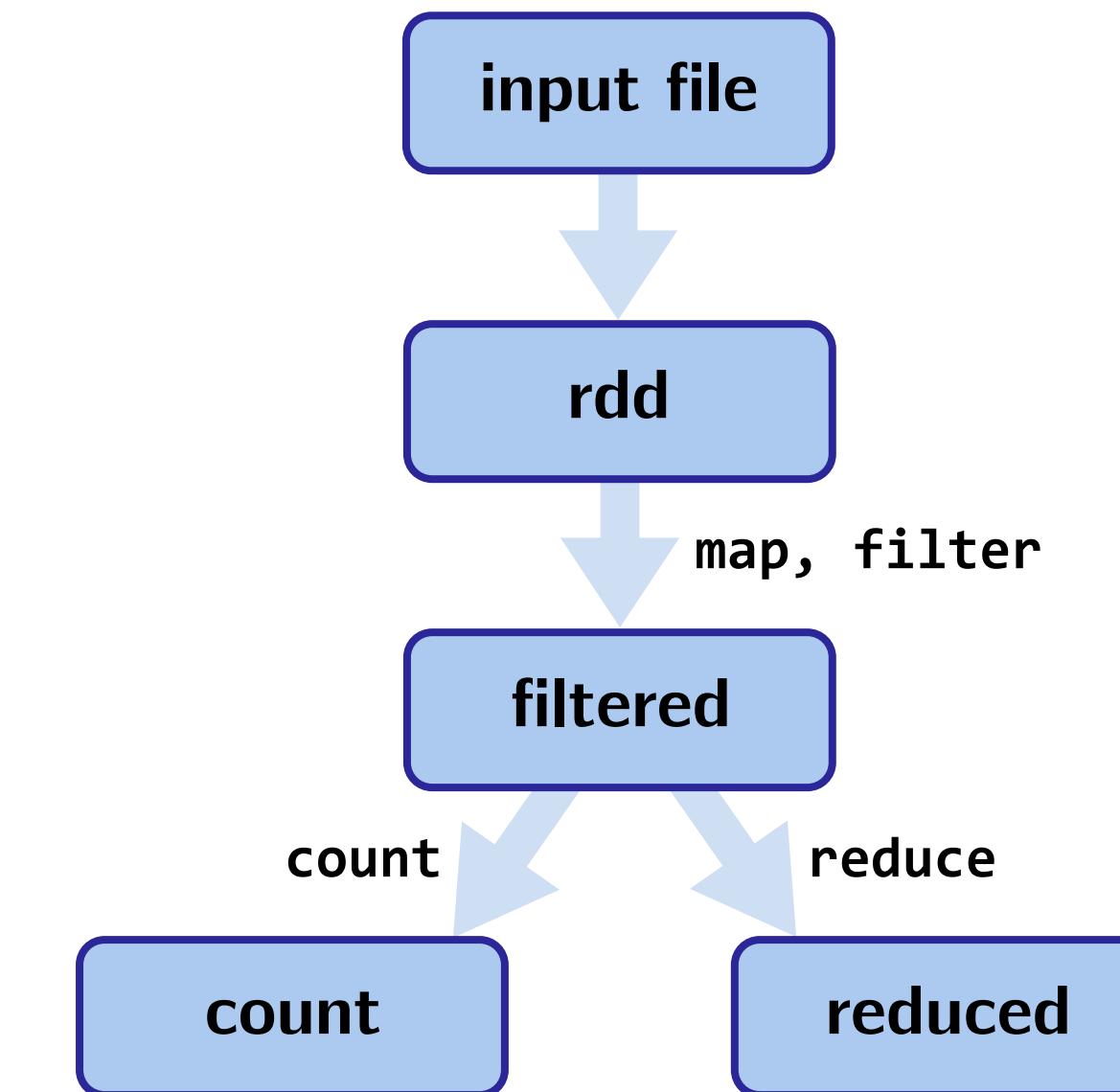
Computations on RDDs are represented as a **lineage graph**; a Directed Acyclic Graph (DAG) representing the computations done on the RDD.

Lineages

Computations on RDDs are represented as a **lineage graph**; a Directed Acyclic Graph (DAG) representing the computations done on the RDD.

Example:

```
val rdd = sc.textFile(...)  
val filtered = rdd.map(...)  
    .filter(...)  
    .persist()  
  
val count = filtered.count()  
val reduced = filtered.reduce(...)
```

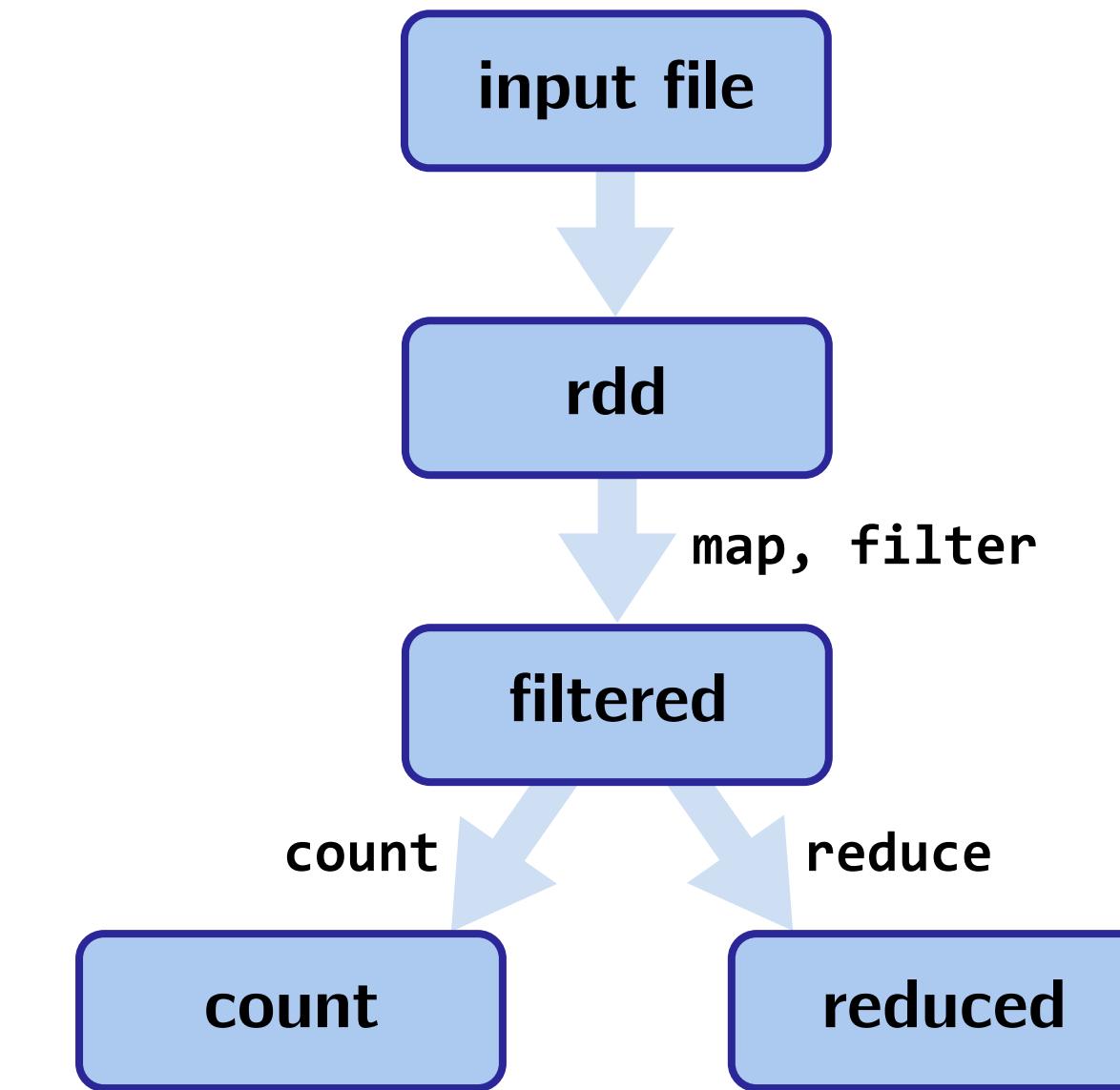


Lineages

Computations on RDDs are represented as a **lineage graph**; a Directed Acyclic Graph (DAG) representing the computations done on the RDD.

Example:

```
val rdd = sc.textFile(...)  
val filtered = rdd.map(...)  
    .filter(...)  
    .persist()  
  
val count = filtered.count()  
val reduced = filtered.reduce(...)
```



Spark represents RDDs in terms of these lineage graphs/DAGs

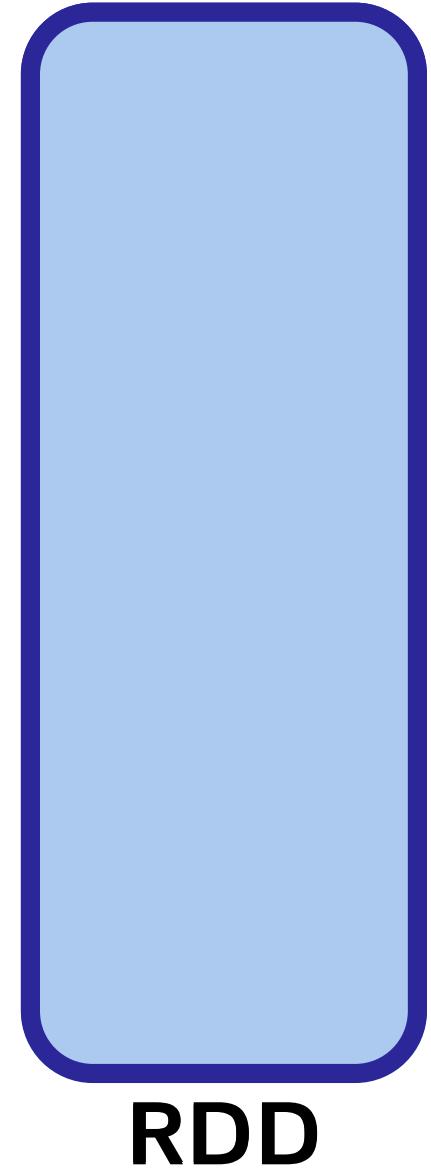
In fact, this is the representation/DAG is what Spark analyzes to do optimizations.

How are RDDs represented?

RDDs are made up of 2 important parts.

(but are made up of 4 parts in total)

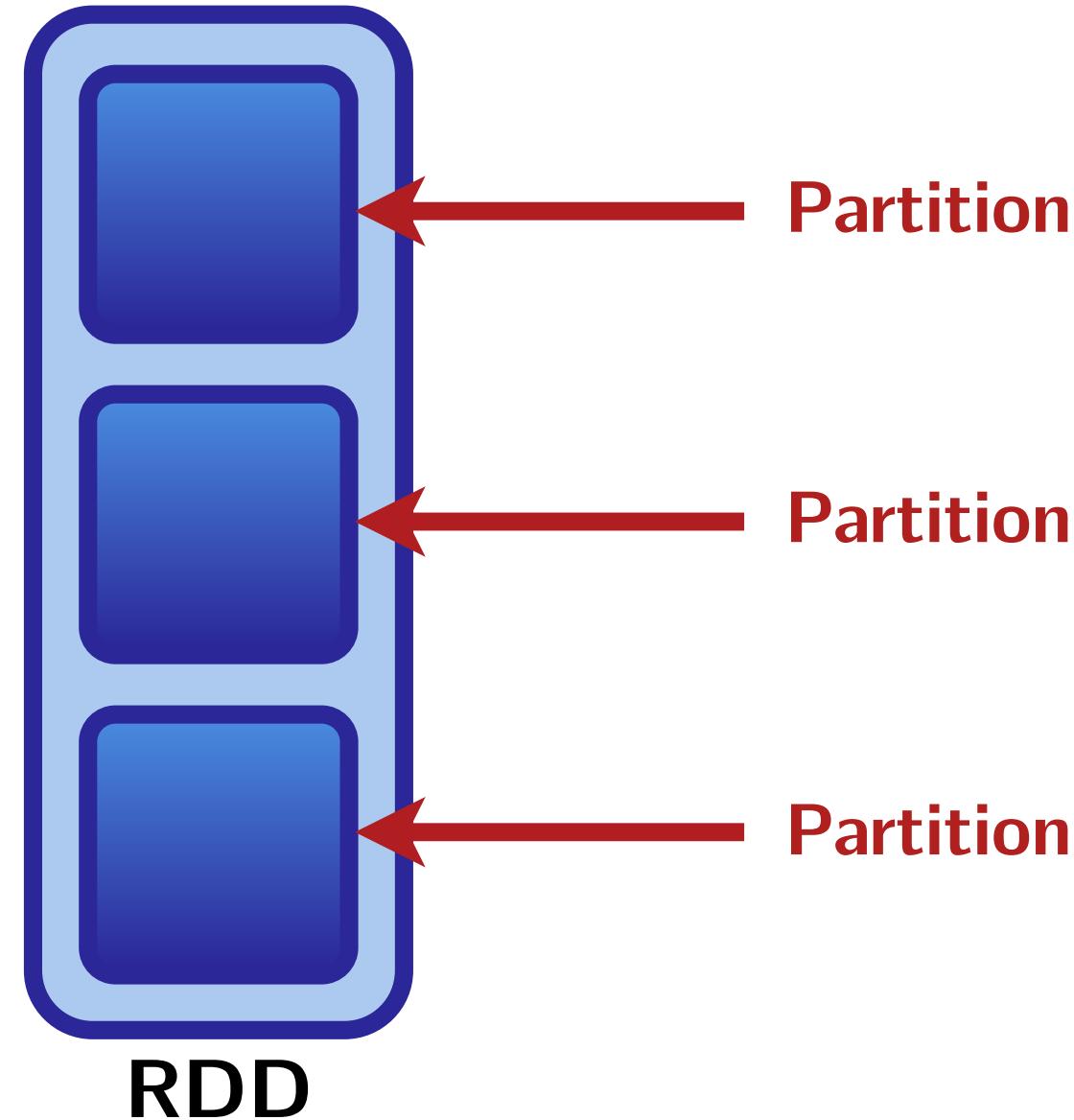
RDDs are represented as:



How are RDDs represented?

RDDs are made up of 2 important parts.

(but are made up of 4 parts in total)



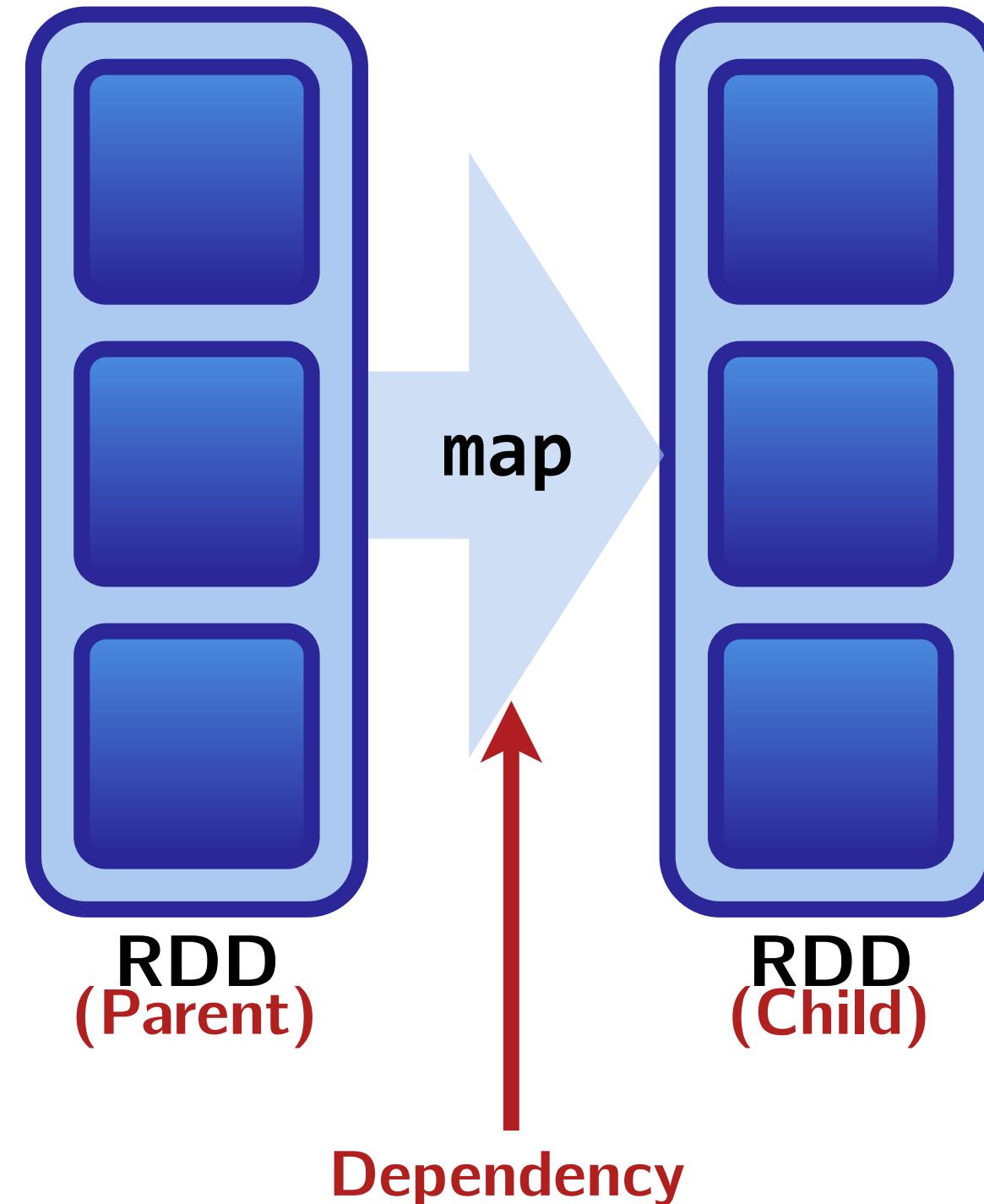
RDDs are represented as:

- ▶ **Partitions.** Atomic pieces of the dataset.
One or many per compute node.

How are RDDs represented?

RDDs are made up of 2 important parts.

(but are made up of 4 parts in total)



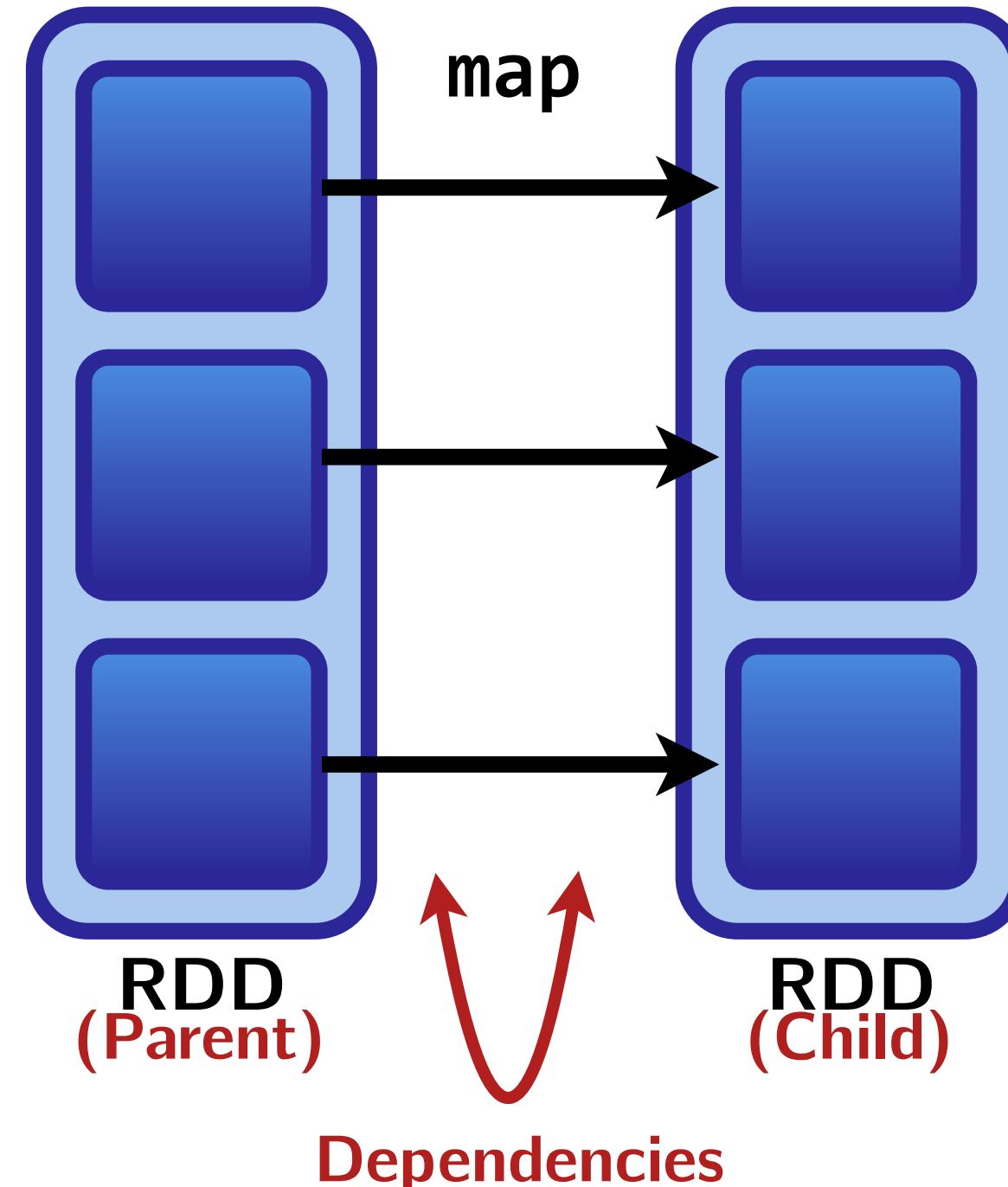
RDDs are represented as:

- ▶ **Partitions.** Atomic pieces of the dataset.
One or many per compute node.
- ▶ **Dependencies.** Models relationship
between this RDD and its partitions
with the RDD(s) it was derived from.

How are RDDs represented?

RDDs are made up of 2 important parts.

(but are made up of 4 parts in total)



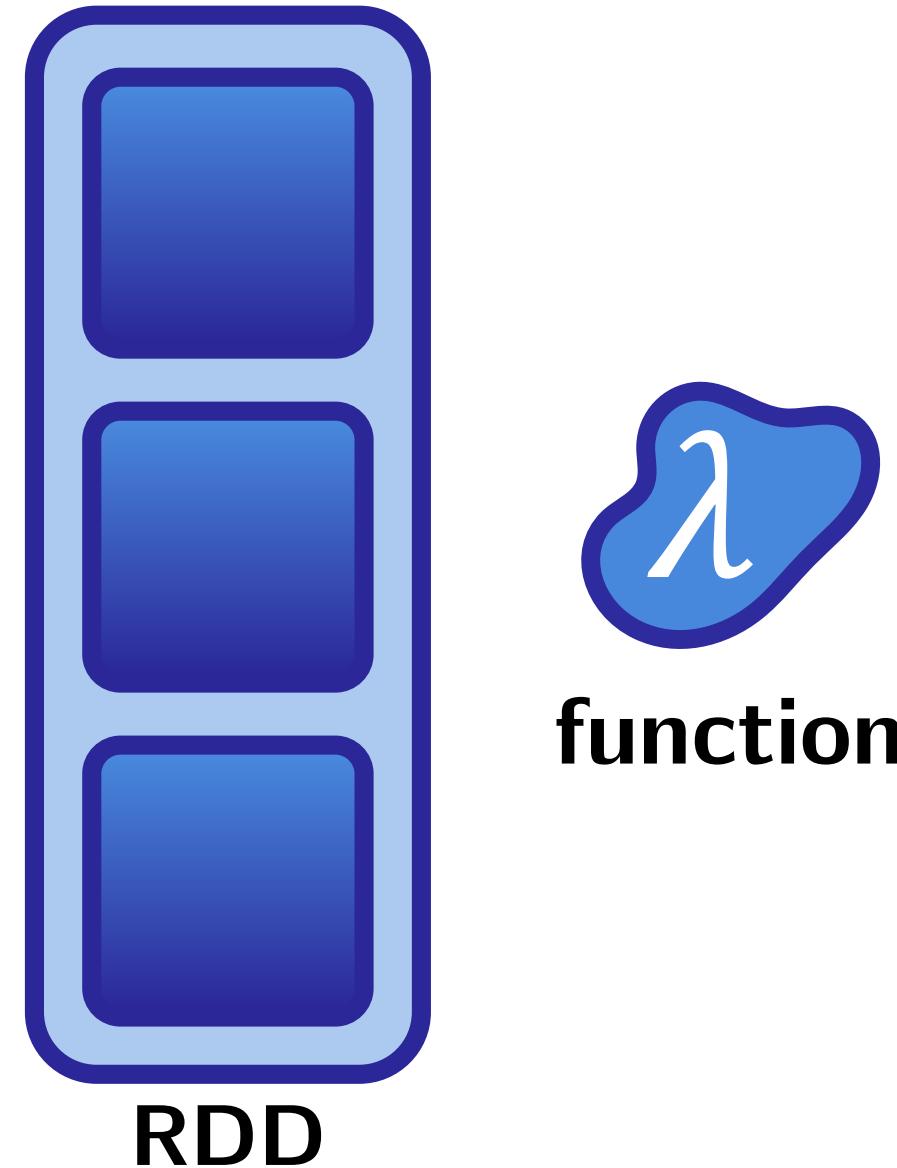
RDDs are represented as:

- ▶ Partitions. Atomic pieces of the dataset.
One or many per compute node.
- ▶ **Dependencies.** Models relationship
between this RDD **and its partitions**
with the RDD(s) it was derived from.

How are RDDs represented?

RDDs are made up of 2 important parts.

(but are made up of 4 parts in total)



RDDs are represented as:

- ▶ **Partitions.** Atomic pieces of the dataset.
One or many per compute node.
- ▶ **Dependencies.** Models relationship
between this RDD and its partitions
with the RDD(s) it was derived from.
- ▶ **A function** for computing the dataset
based on its parent RDDs.
- ▶ **Metadata** about its partitioning scheme
and data placement.

RDD Dependencies and Shuffles

Previously, we arrived at the following rule of thumb for trying to determine when a shuffle might occur:

Rule of thumb: a shuffle *can* occur when the resulting RDD depends on other elements from the same RDD or another RDD.

RDD Dependencies and Shuffles

Previously, we arrived at the following rule of thumb for trying to determine when a shuffle might occur:

Rule of thumb: a shuffle *can* occur when the resulting RDD depends on other elements from the same RDD or another RDD.

In fact, RDD dependencies encode when data must move across the network.

RDD Dependencies and Shuffles

Previously, we arrived at the following rule of thumb for trying to determine when a shuffle might occur:

Rule of thumb: a shuffle *can* occur when the resulting RDD depends on other elements from the same RDD or another RDD.

In fact, RDD dependencies encode when data must move across the network.

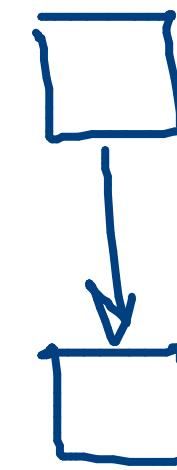
Transformations cause shuffles. Transformations can have two kinds of dependencies:

1. **Narrow Dependencies**
2. **Wide Dependencies**

Narrow Dependencies vs Wide Dependencies

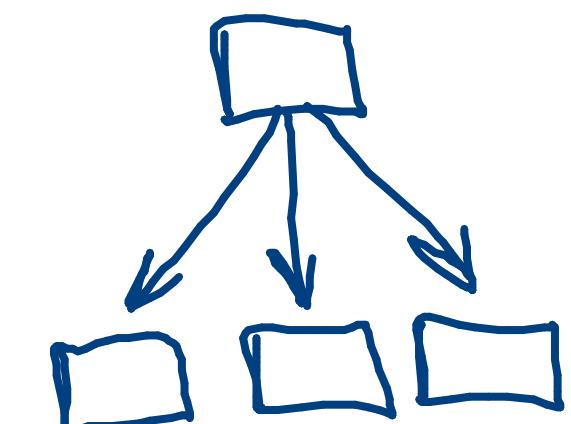
Narrow Dependencies

Each partition of the parent RDD is used by at most one partition of the child RDD.



Wide Dependencies

Each partition of the parent RDD may be depended on by **multiple** child partitions.



Narrow Dependencies vs Wide Dependencies

Narrow Dependencies

Each partition of the parent RDD is used by at most one partition of the child RDD.

Fast! No shuffle necessary. Optimizations like pipelining possible.

Wide Dependencies

Each partition of the parent RDD may be depended on by **multiple** child partitions.

Slow! Requires all or some data to be shuffled over the network.

Narrow Dependencies vs Wide Dependencies, Visually

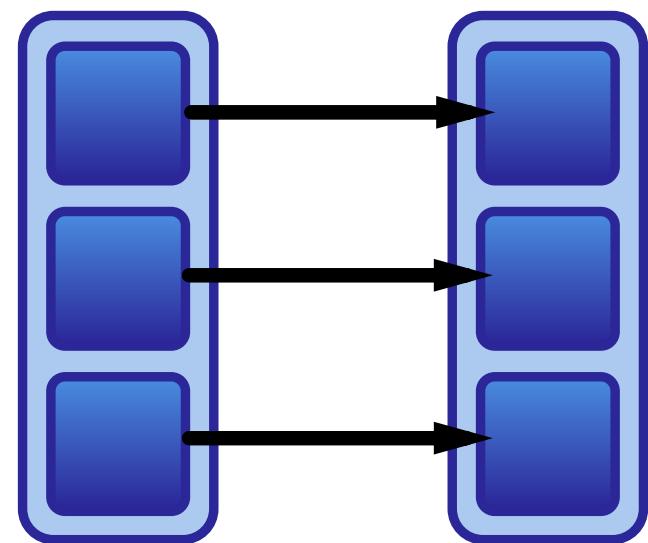
Narrow dependencies:

Each partition of the parent RDD is used by at most one partition of the child RDD.

Narrow Dependencies vs Wide Dependencies, Visually

Narrow dependencies:

Each partition of the parent RDD is used by at most one partition of the child RDD.

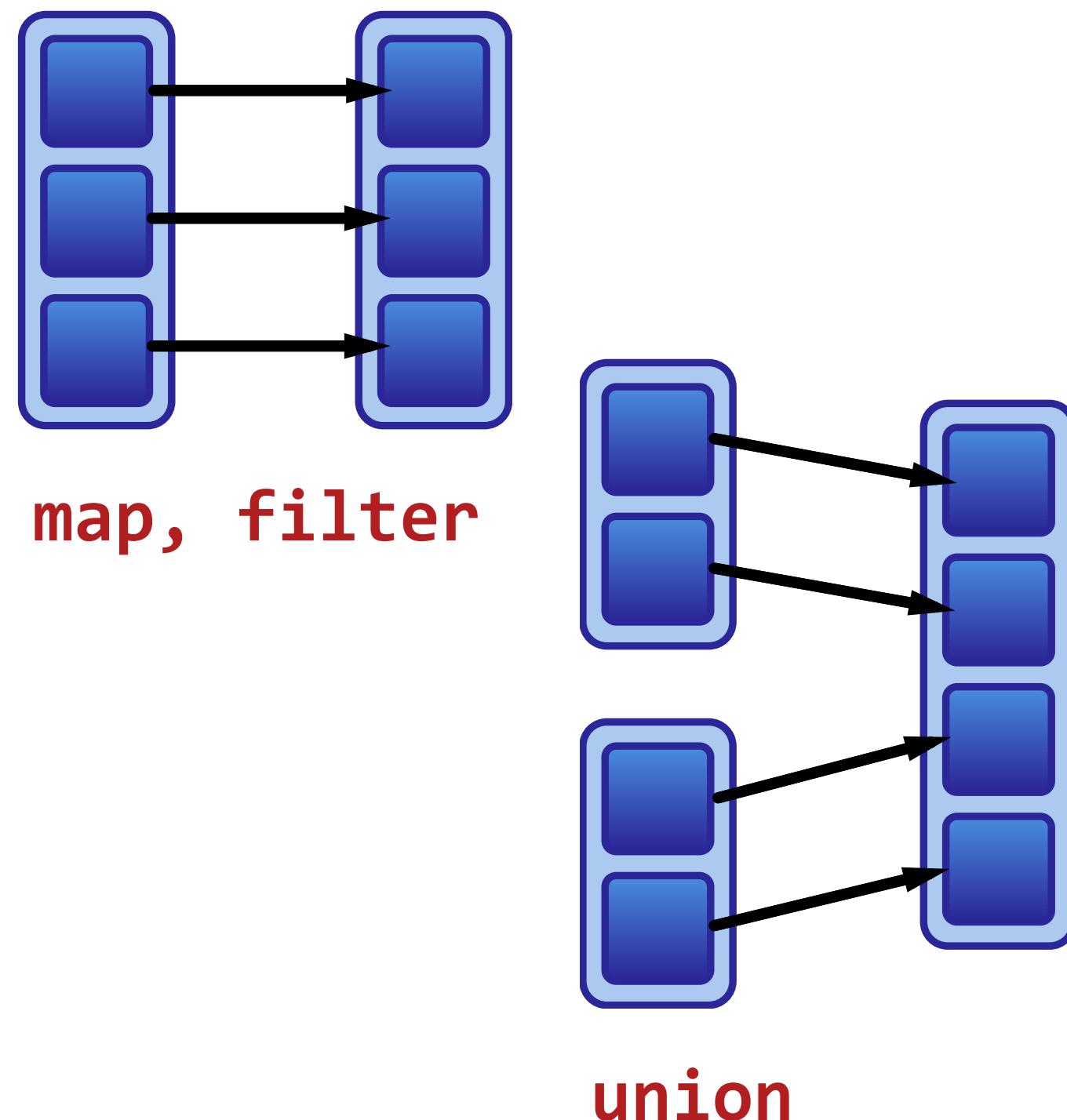


map, filter

Narrow Dependencies vs Wide Dependencies, Visually

Narrow dependencies:

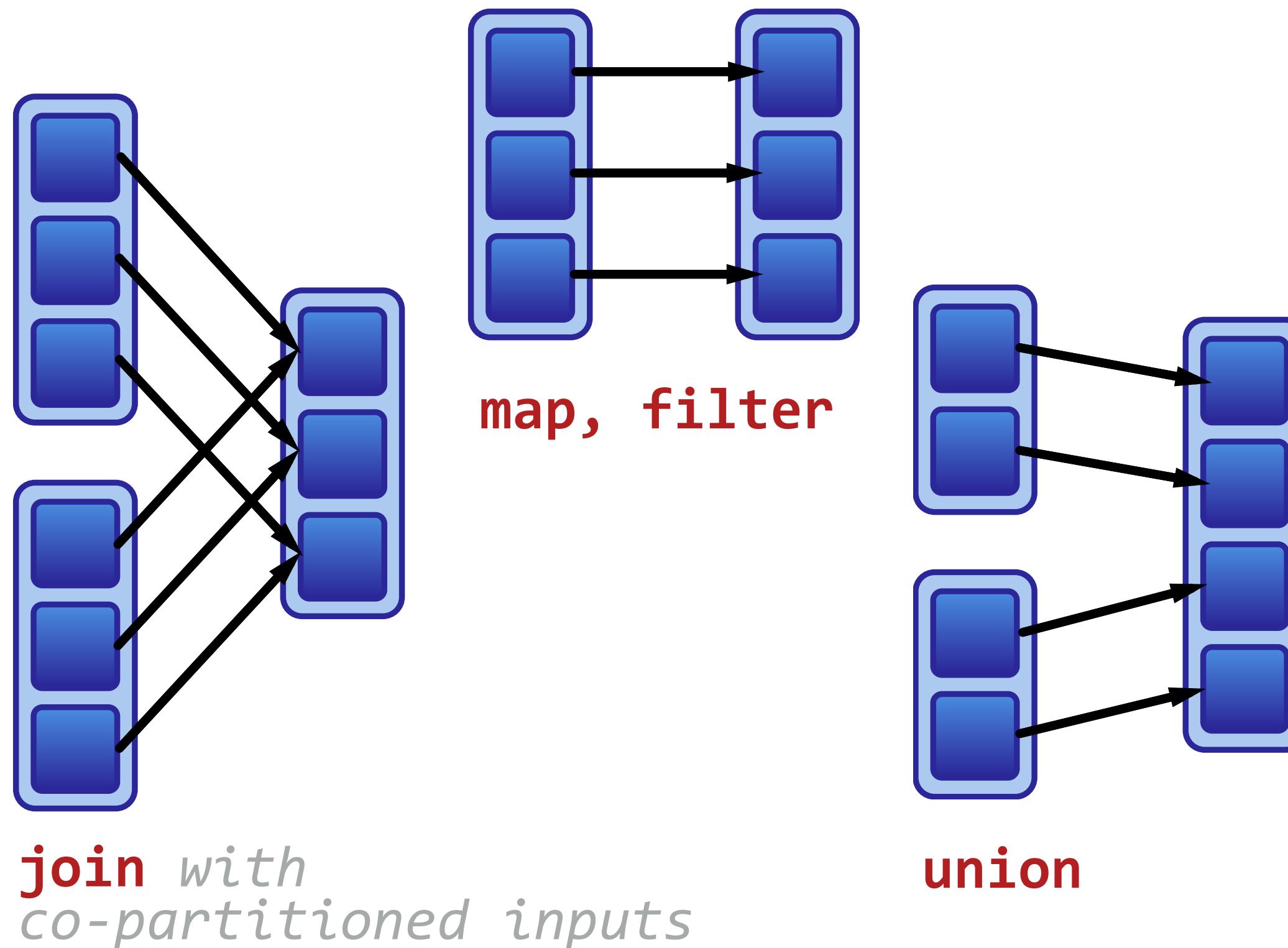
Each partition of the parent RDD is used by at most one partition of the child RDD.



Narrow Dependencies vs Wide Dependencies, Visually

Narrow dependencies:

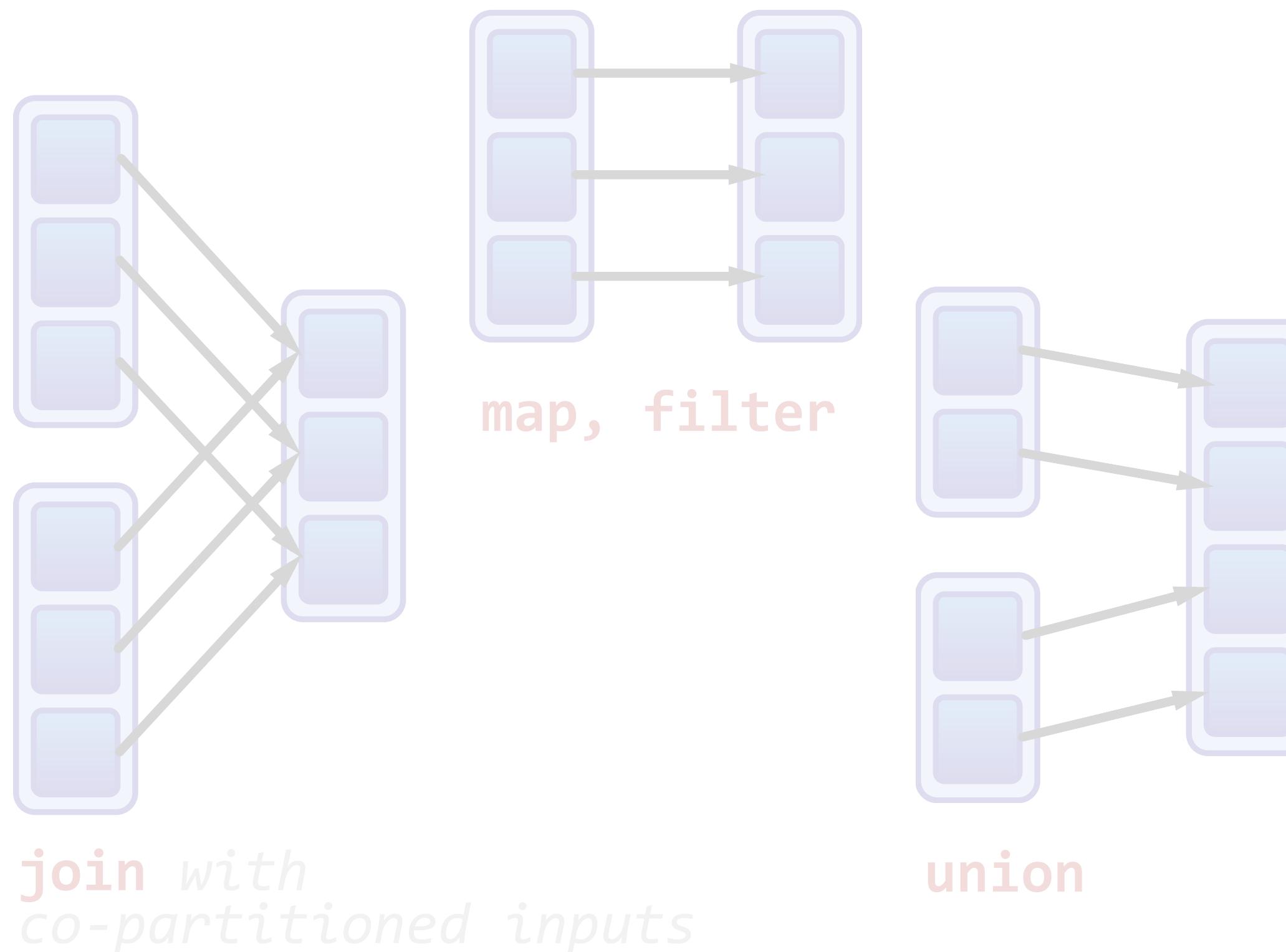
Each partition of the parent RDD is used by at most one partition of the child RDD.



Narrow Dependencies vs Wide Dependencies, Visually

Narrow dependencies:

Each partition of the parent RDD is used by at most one partition of the child RDD.



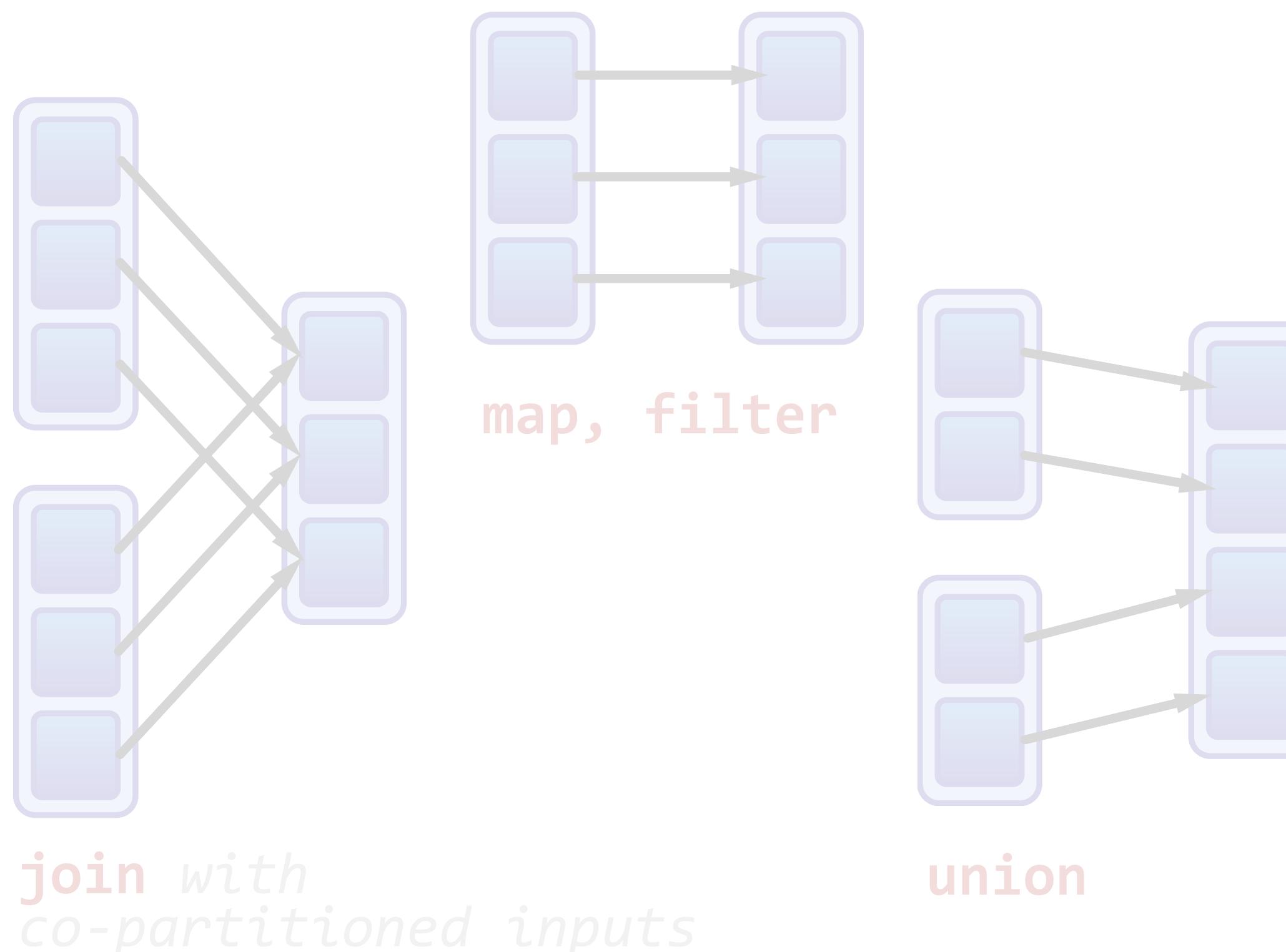
Wide dependencies:

Each partition of the parent RDD may be depended on by multiple child partitions.

Narrow Dependencies vs Wide Dependencies, Visually

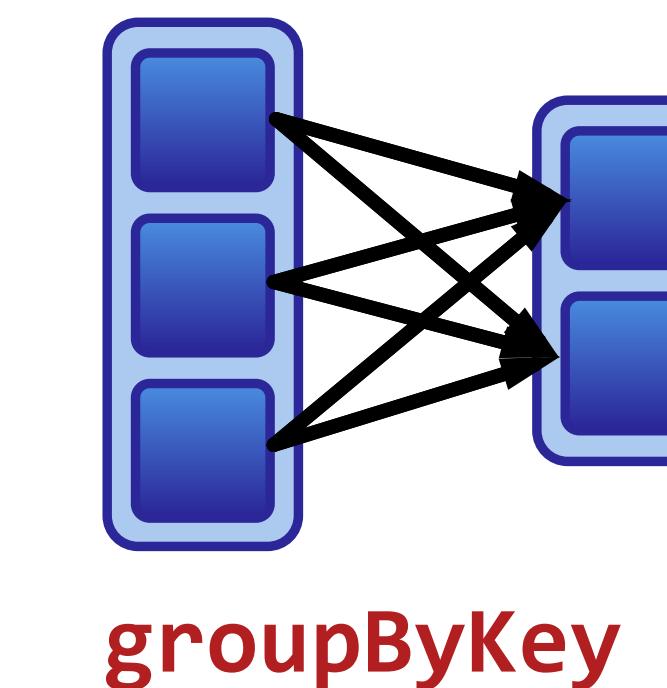
Narrow dependencies:

Each partition of the parent RDD is used by at most one partition of the child RDD.



Wide dependencies:

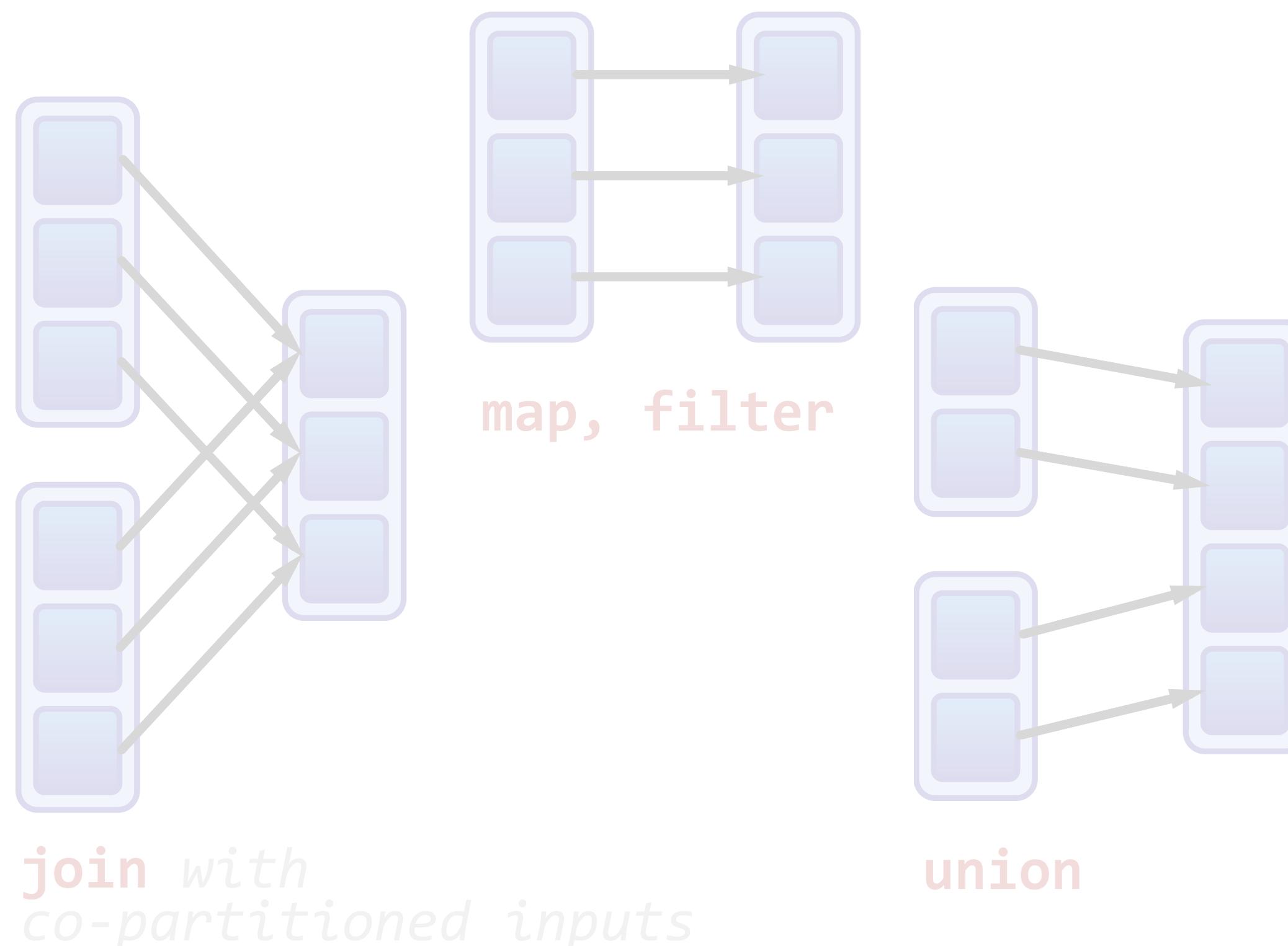
Each partition of the parent RDD may be depended on by multiple child partitions.



Narrow Dependencies vs Wide Dependencies, Visually

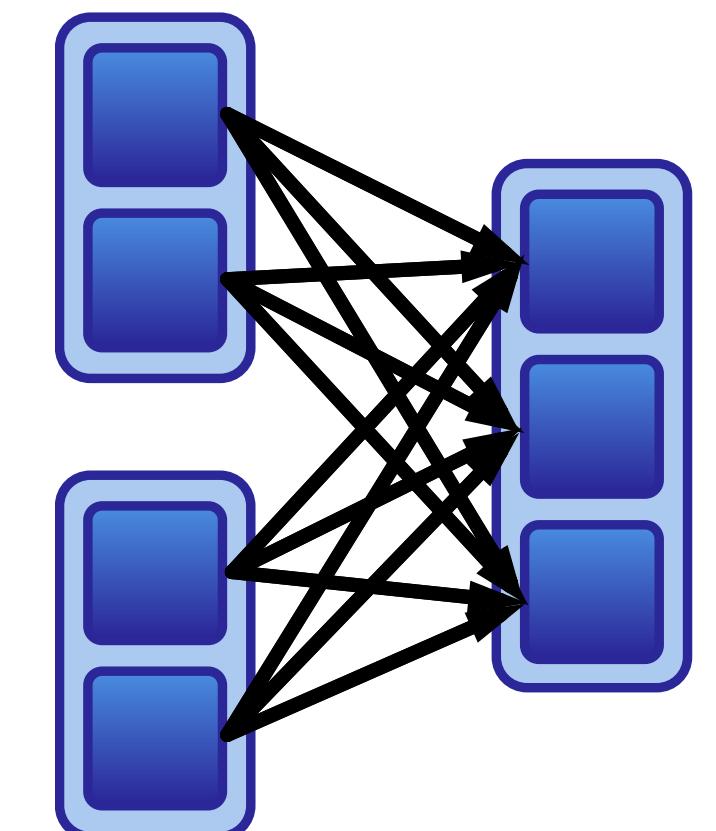
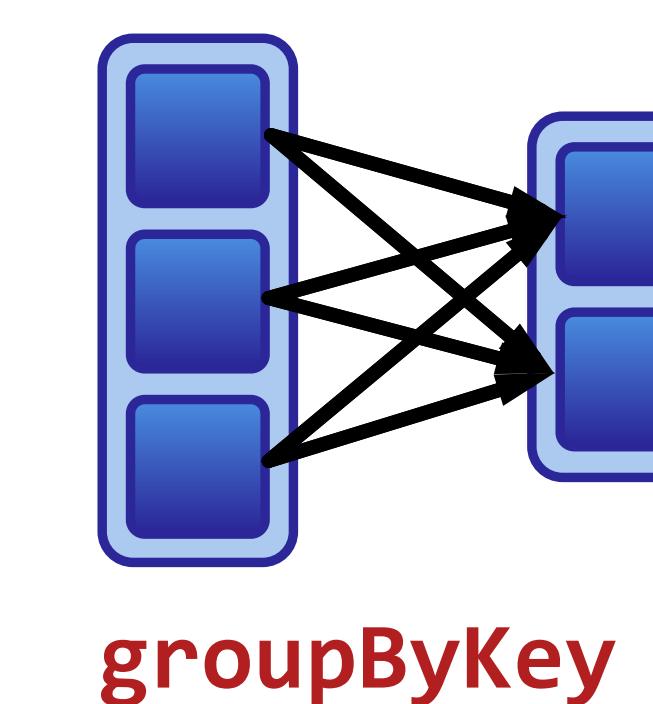
Narrow dependencies:

Each partition of the parent RDD is used by at most one partition of the child RDD.



Wide dependencies:

Each partition of the parent RDD may be depended on by multiple child partitions.

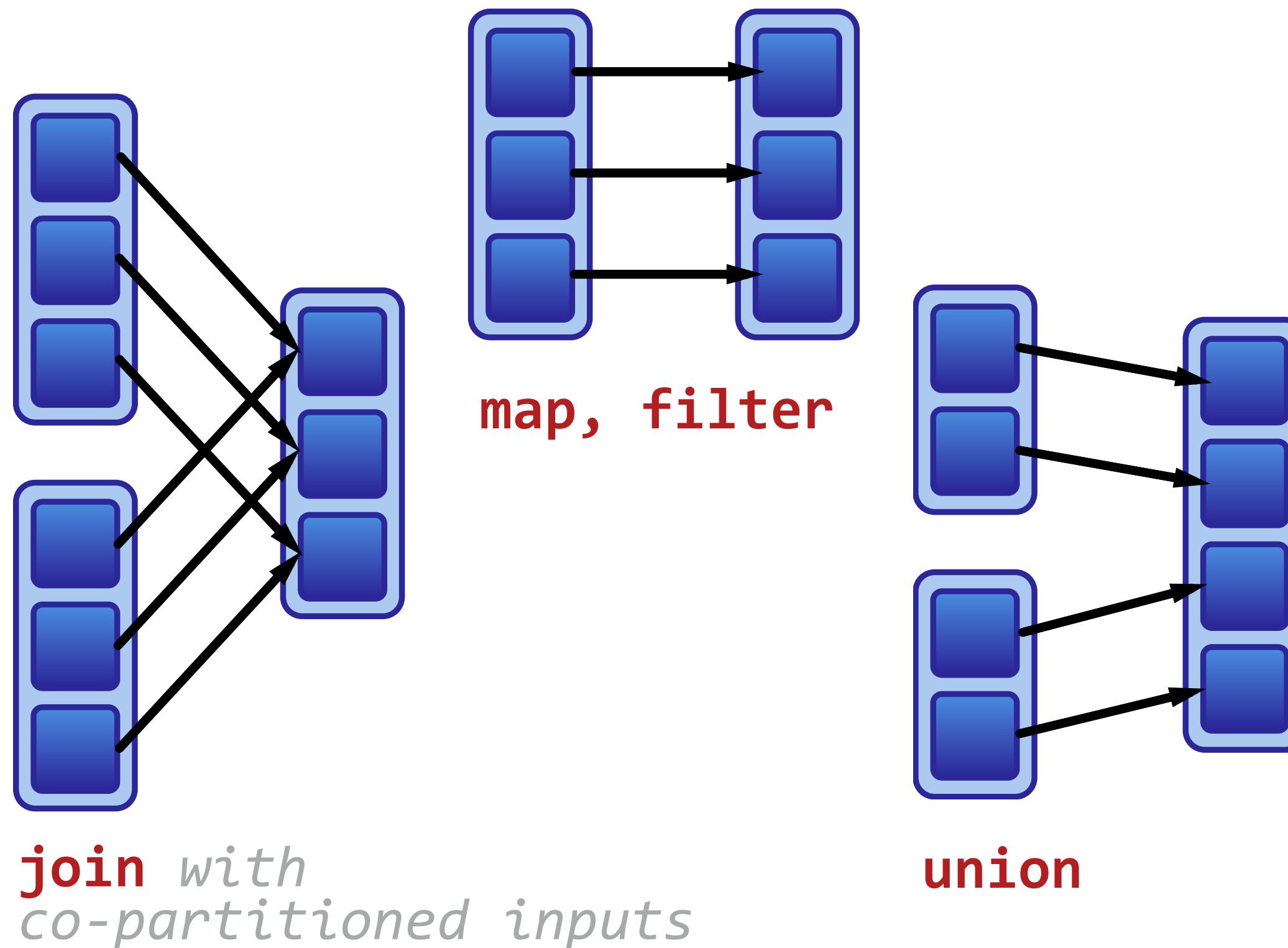


join
with
inputs not
co-partitioned

Narrow Dependencies vs Wide Dependencies, Visually

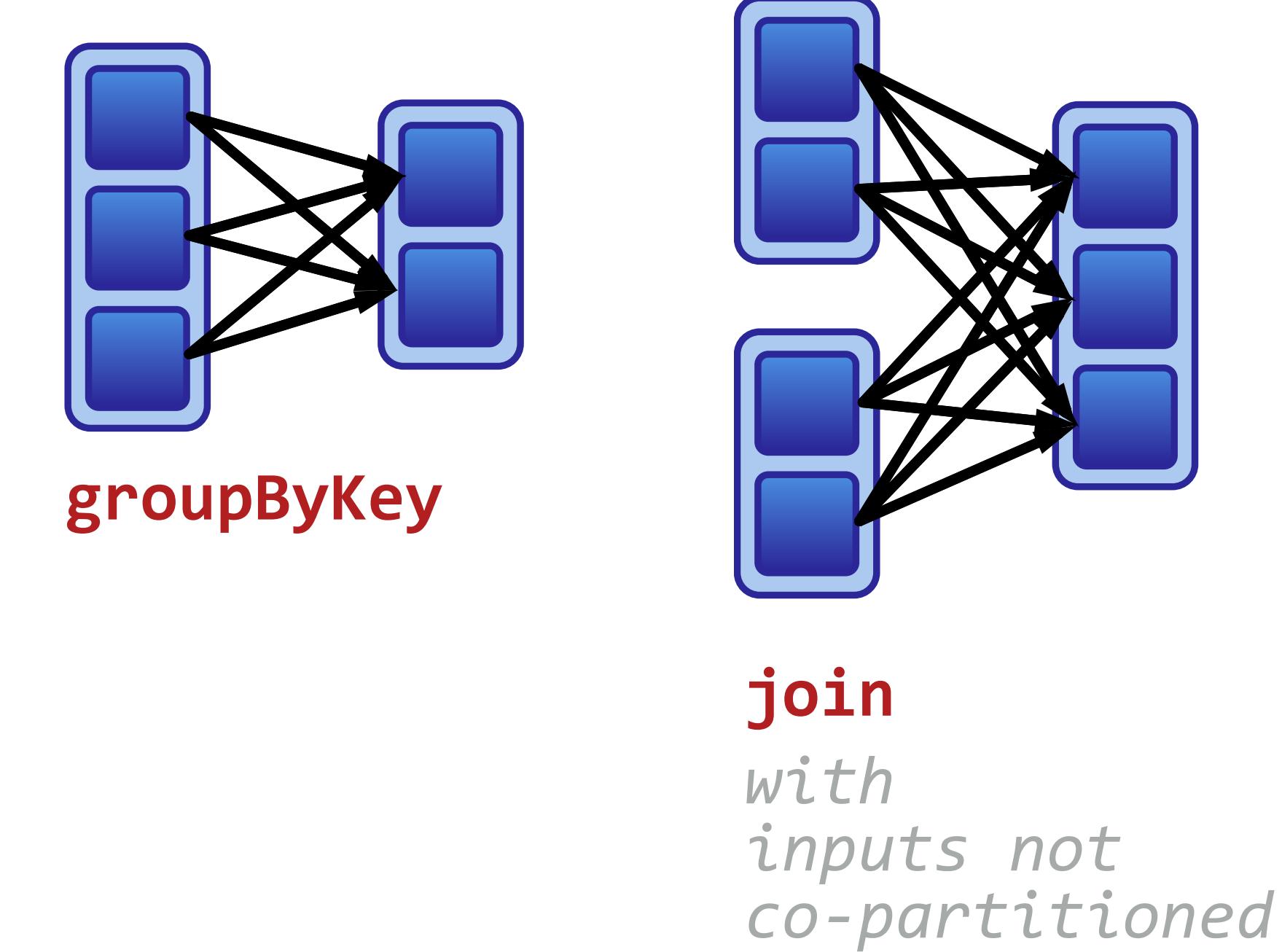
Narrow dependencies:

Each partition of the parent RDD is used by at most one partition of the child RDD.



Wide dependencies:

Each partition of the parent RDD may be depended on by multiple child partitions.



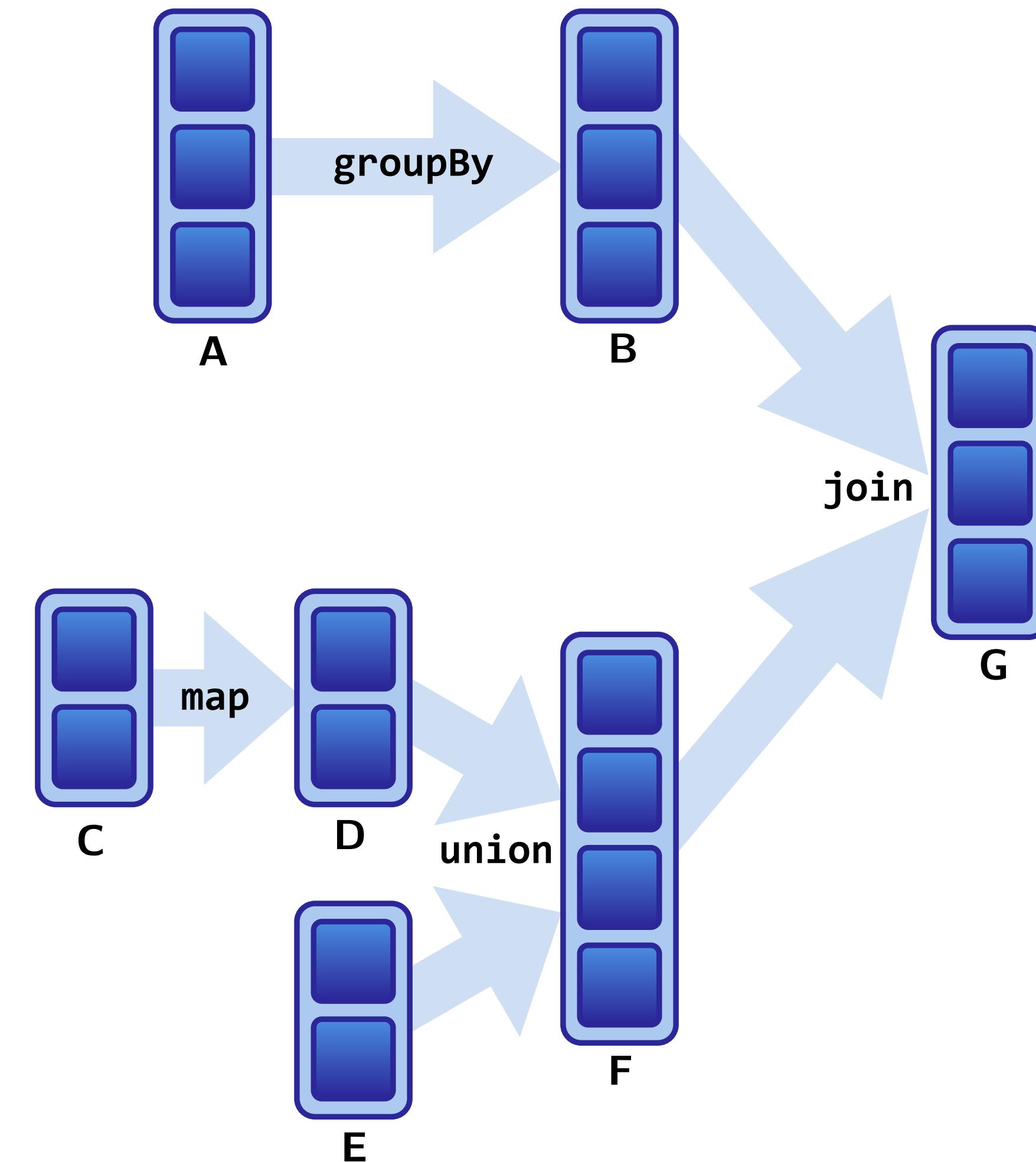
Narrow Dependencies vs Wide Dependencies, Visually

Let's visualize an example program and its dependencies.

Narrow Dependencies vs Wide Dependencies, Visually

Let's visualize an example program and its dependencies.

Conceptually assuming the DAG:



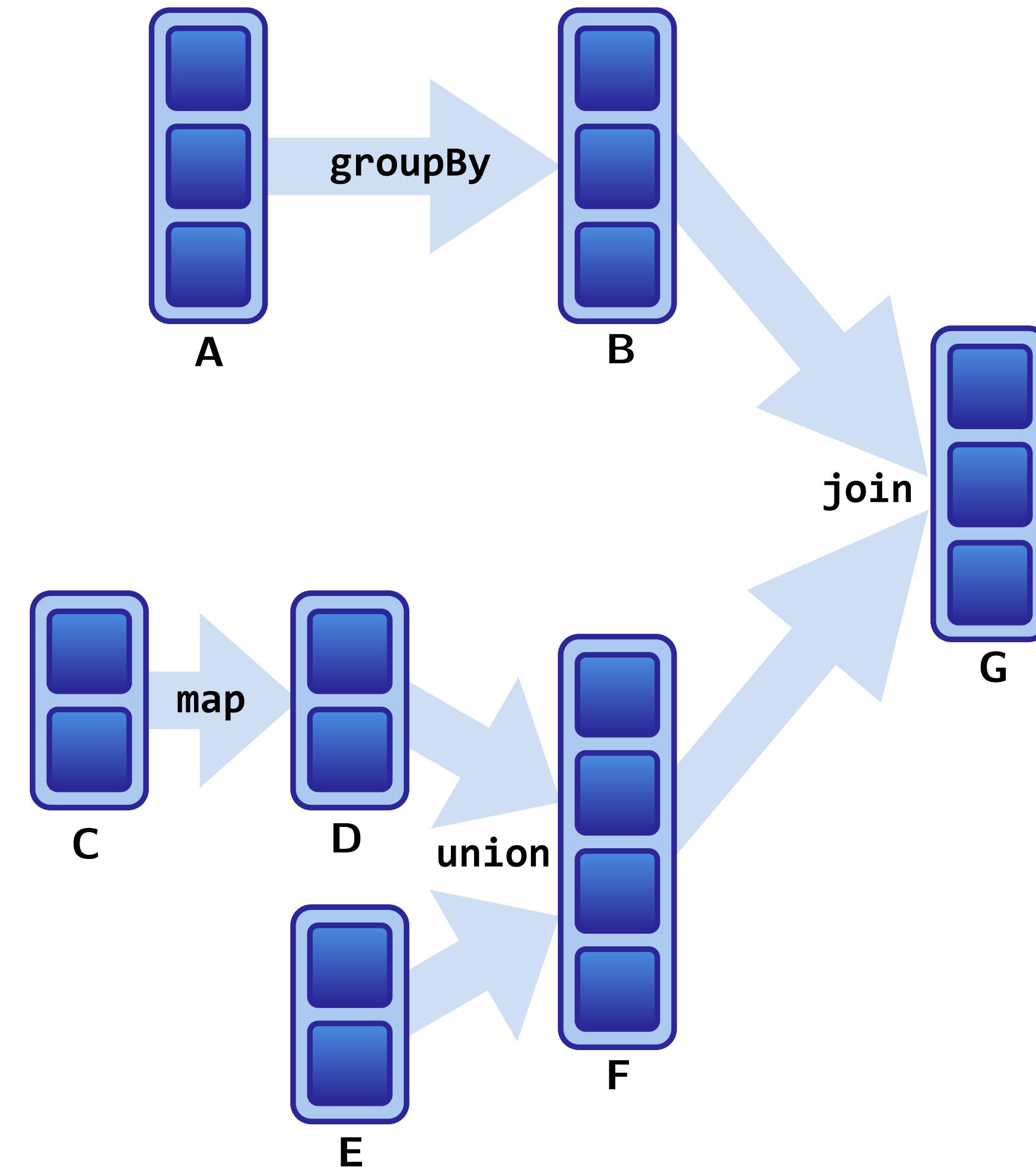
Narrow Dependencies vs Wide Dependencies, Visually

Let's visualize an example program and its dependencies.

Conceptually assuming the DAG:

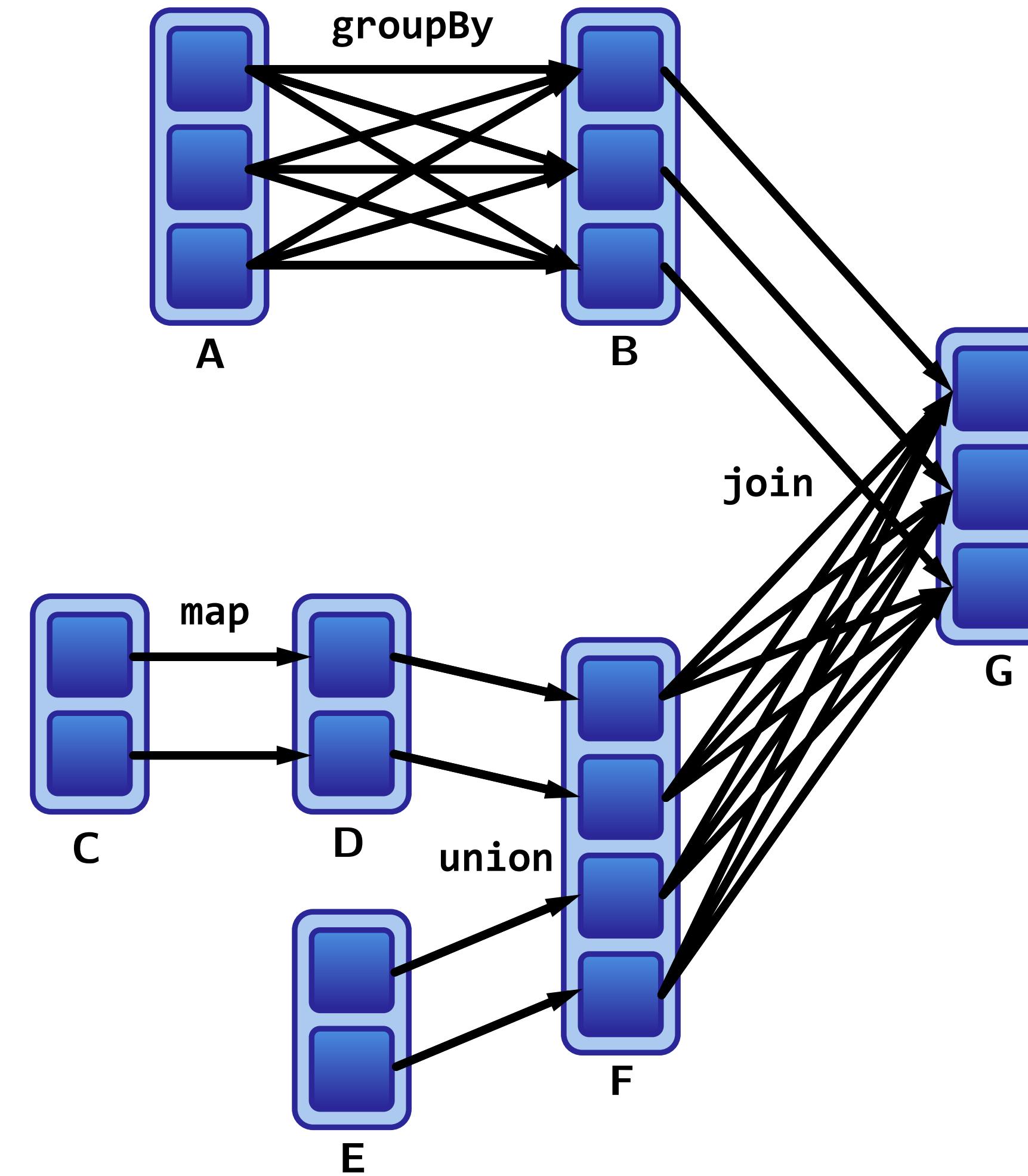
What do the dependencies look like?

Which dependencies are wide, and which are narrow?



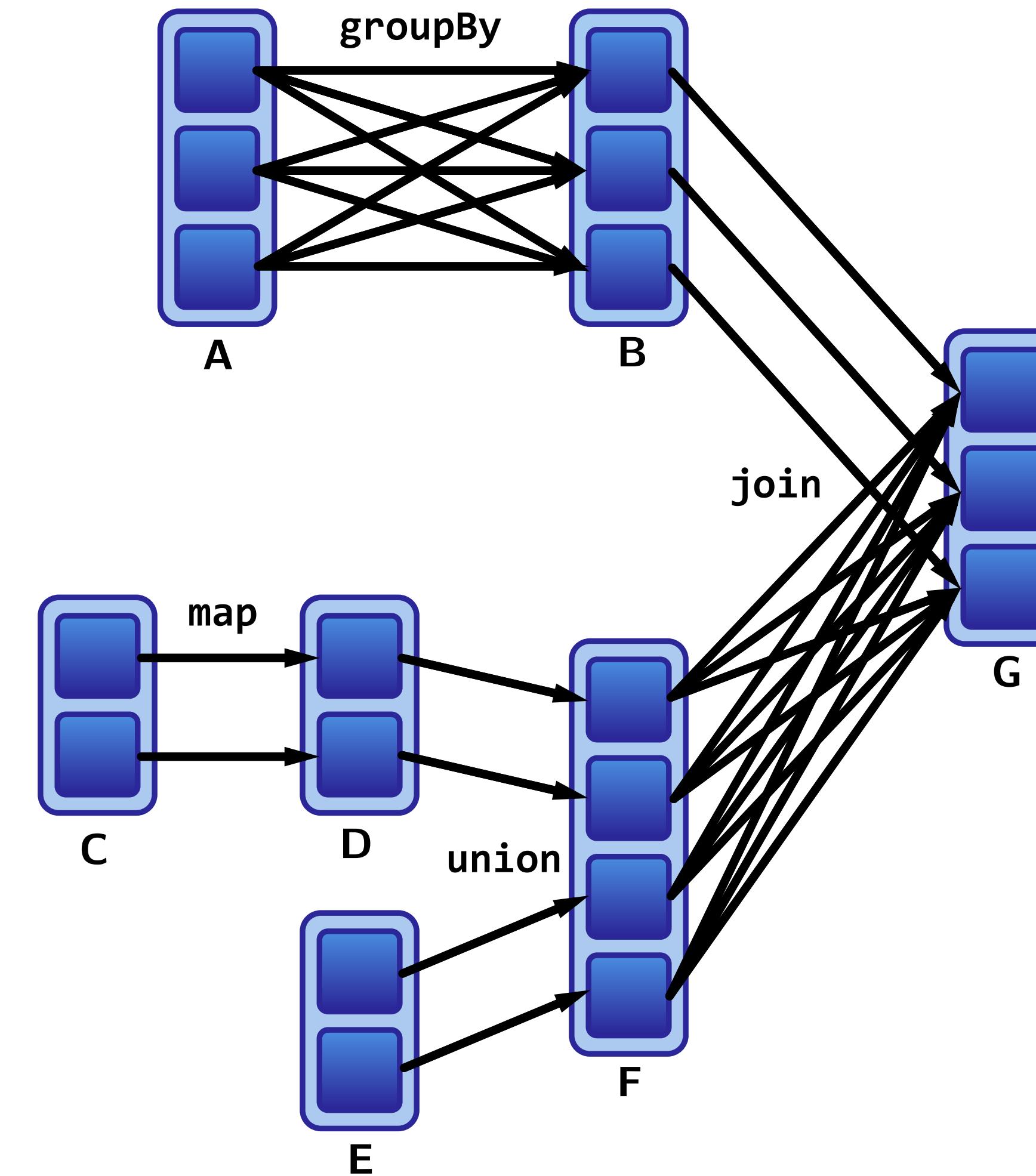
Narrow Dependencies vs Wide Dependencies, Visually

Let's visualize an example program and its dependencies.



Narrow Dependencies vs Wide Dependencies, Visually

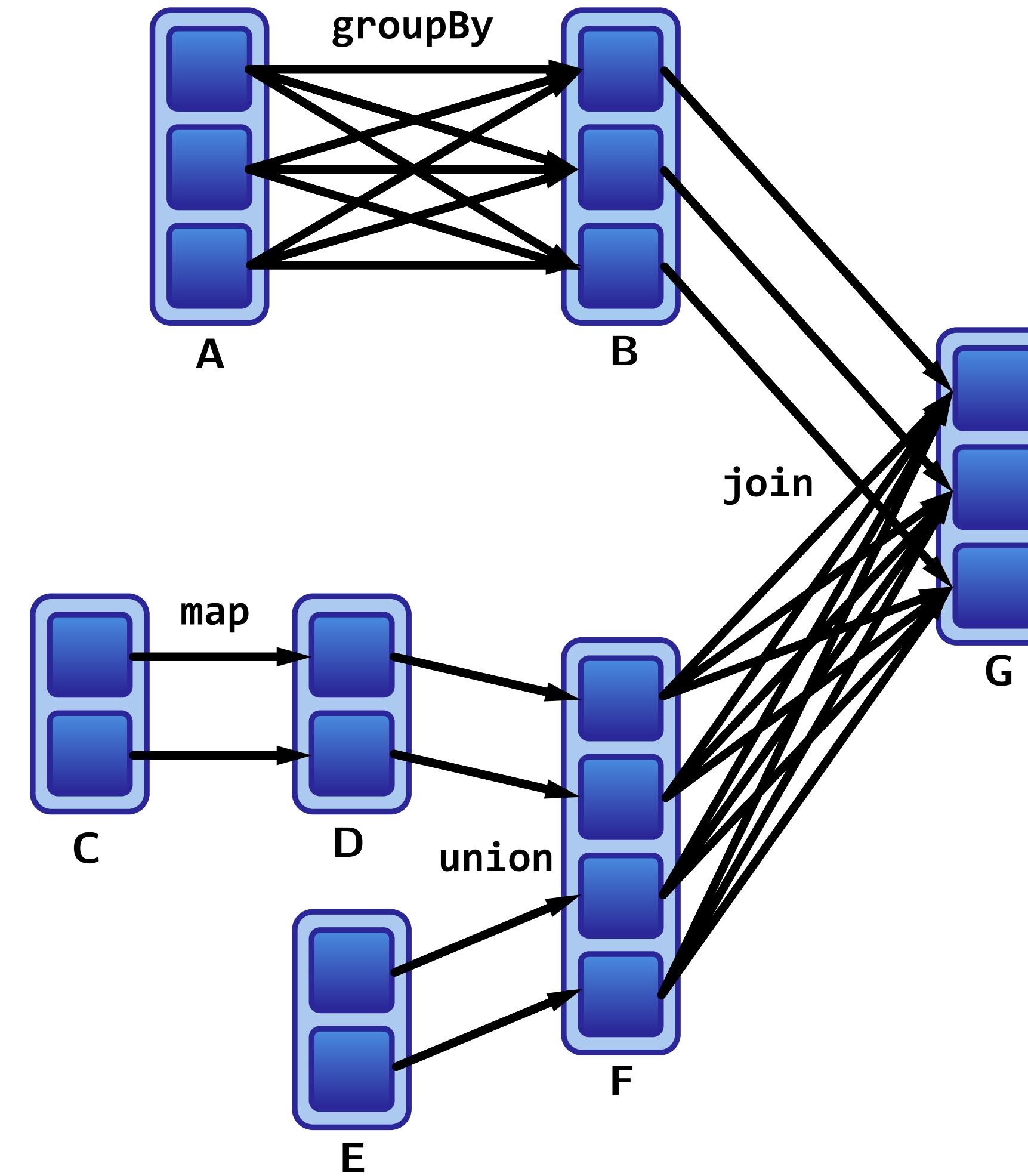
Let's visualize an example program and its dependencies.



Which dependencies are wide, and which are narrow?

Narrow Dependencies vs Wide Dependencies, Visually

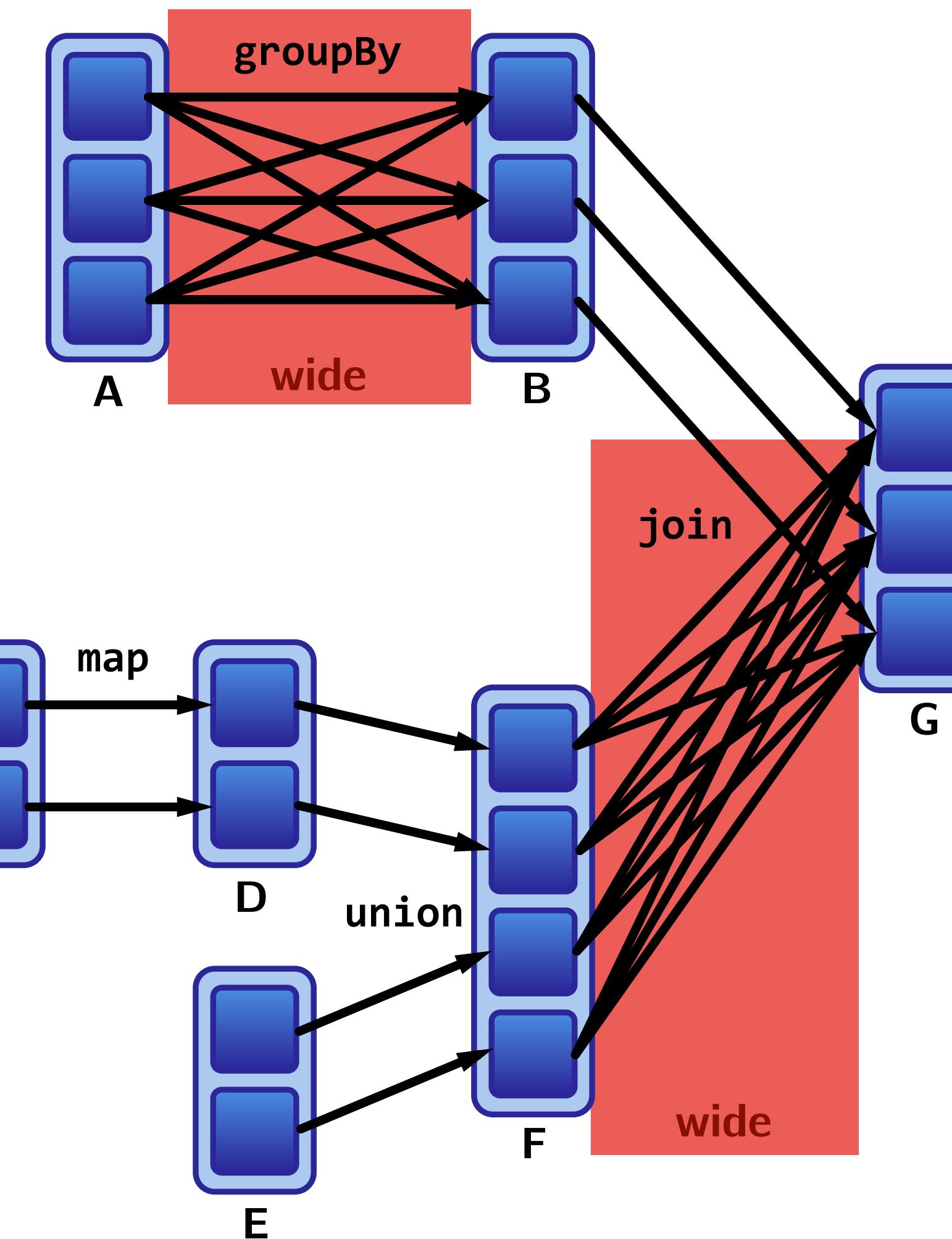
Let's visualize an example program and its dependencies.



Narrow Dependencies vs Wide Dependencies, Visually

Let's visualize an example program and its dependencies.

Wide transformations:
groupBy, join

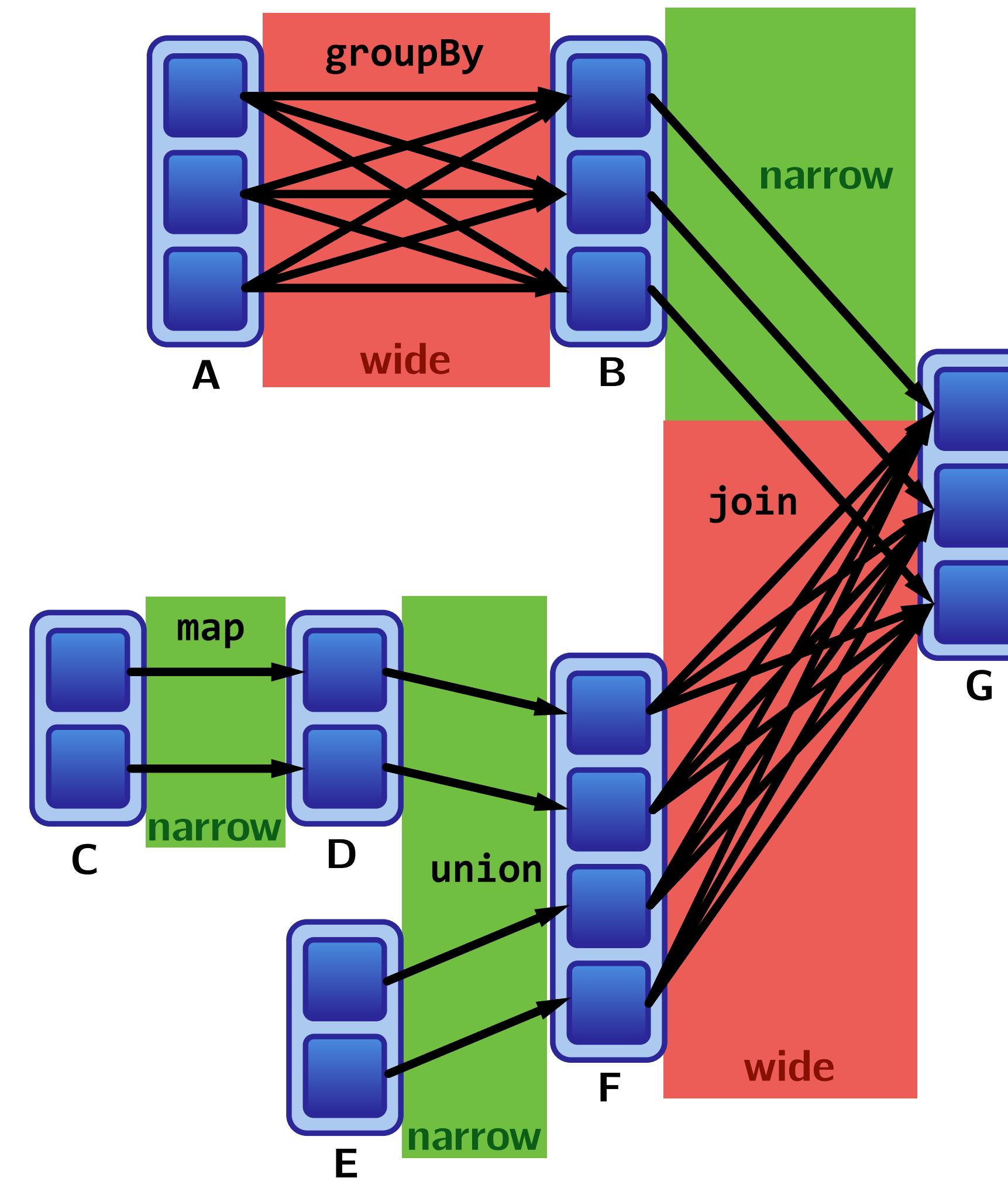


Narrow Dependencies vs Wide Dependencies, Visually

Let's visualize an example program and its dependencies.

Wide transformations:
groupBy, join

Narrow transformations:
map, union, join

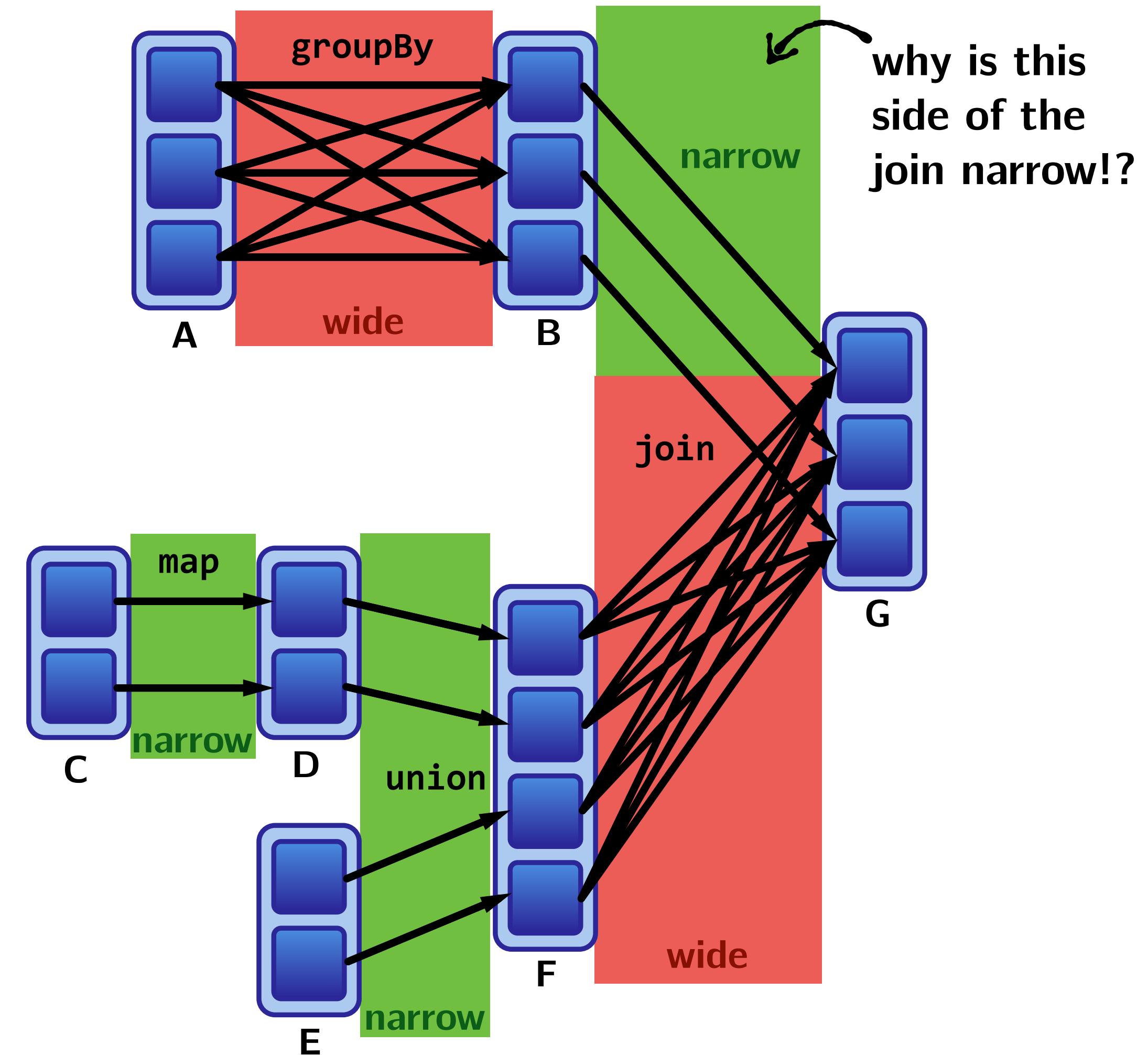


Narrow Dependencies vs Wide Dependencies, Visually

Let's visualize an example program and its dependencies.

Wide transformations:
groupBy, join

Narrow transformations:
map, union, join

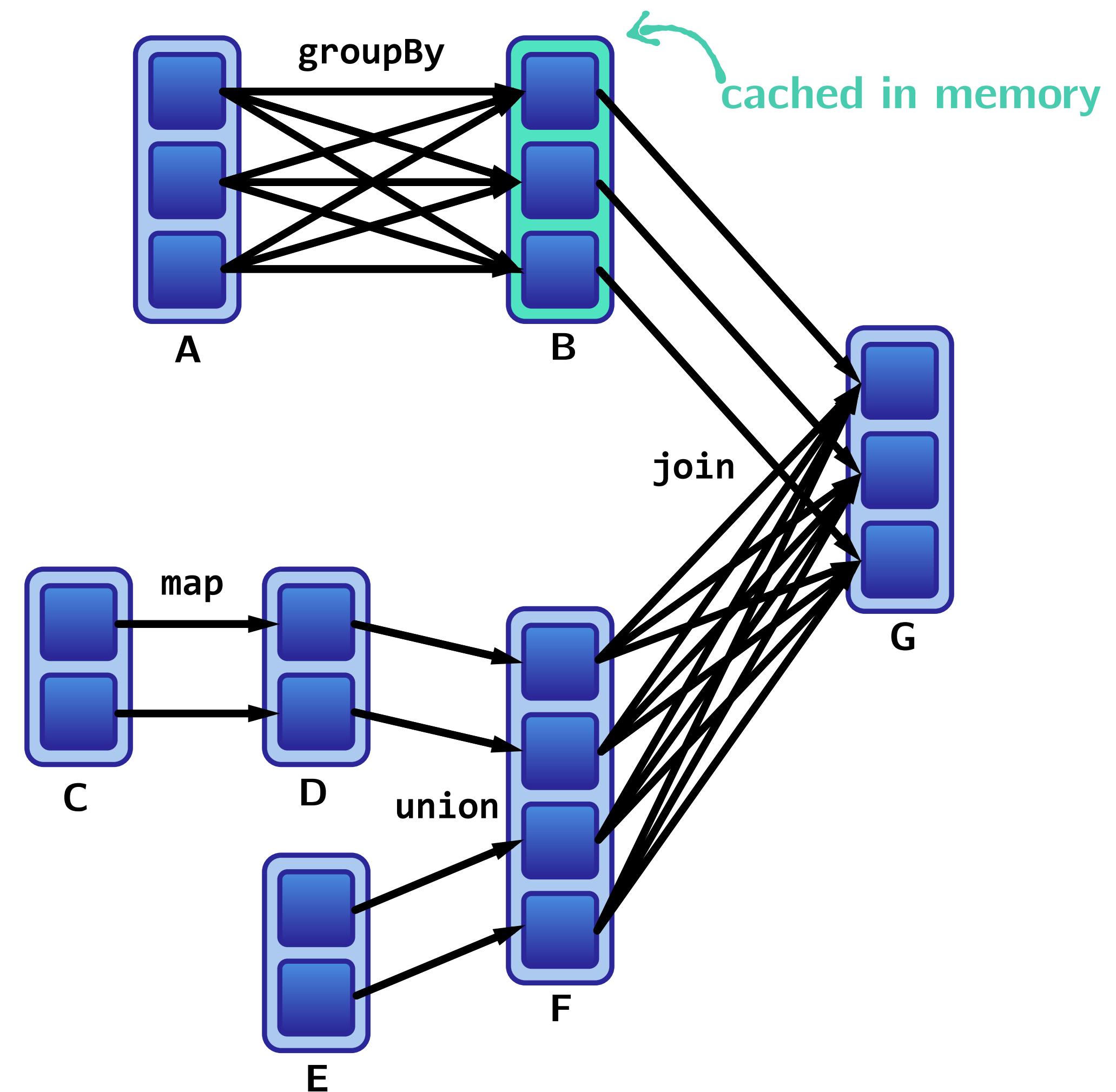


Narrow Dependencies vs Wide Dependencies, Visually

Let's visualize an example program and its dependencies.

Since **G** would be derived from **B**, which itself is derived from a **groupBy** and a shuffle on **A**, you could imagine that we will have already co-partitioned and cached **B** in memory following the call to **groupBy**.

Part of this join is thus a narrow transformation.



Which transformations have which kind of dependency?

Transformations with narrow dependencies:

- map
- mapValues
- flatMap
- filter
- mapPartitions
- mapPartitionsWithIndex

Transformations with wide dependencies:

(might cause a shuffle)

- cogroup
- groupWith
- join
- leftOuterJoin
- rightOuterJoin
- groupByKey
- reduceByKey
- combineByKey
- distinct
- intersection
- repartition
- coalesce

How can I find out?

dependencies method on RDDs.

`dependencies` returns a sequence of Dependency objects, which are actually the dependencies used by Spark's scheduler to know how this RDD depends on other RDDs.

The sorts of dependency objects the `dependencies` method may return include:

Narrow dependency objects:

- ▶ `OneToOneDependency`
- ▶ `PruneDependency`
- ▶ `RangeDependency`

Wide dependency objects:

- ▶ `ShuffleDependency`

How can I find out?

dependencies method on RDDs.

dependencies returns a sequence of Dependency objects, which are actually the dependencies used by Spark's scheduler to know how this RDD depends on other RDDs.

```
val wordsRdd = sc.parallelize(largeList)
val pairs = wordsRdd.map(c => (c, 1))
              .groupByKey()
              .dependencies
// pairs: Seq[org.apache.spark.Dependency[_]] =
// List(org.apache.spark.ShuffleDependency@4294a23d)
```

How can I find out?

toDebugString method on RDDs.

`toDebugString` prints out a visualization of the RDD's lineage, and other information pertinent to scheduling. For example, indentations in the output separate groups of narrow transformations that may be pipelined together with wide transformations that require shuffles. These groupings are called *stages*.

```
val wordsRdd = sc.parallelize(largeList)
val pairs = wordsRdd.map(c => (c, 1))
    .groupByKey()
    .toDebugString

//pairs: String =
//(8) ShuffledRDD[219] at groupByKey at <console>:38 []
// +- (8) MapPartitionsRDD[218] at map at <console>:37 []
//     | ParallelCollectionRDD[217] at parallelize at <console>:36 []
```

Lineages and Fault Tolerance

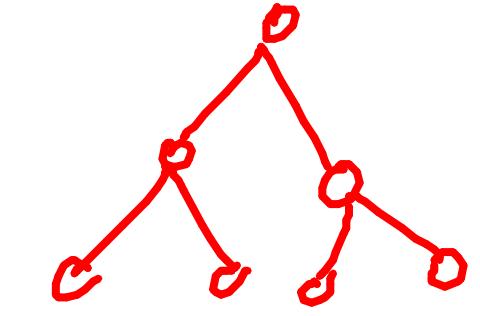
Lineages graphs are the key to fault tolerance in Spark.

Lineages and Fault Tolerance

Lineages graphs are the key to fault tolerance in Spark.

Ideas from **functional programming** enable fault tolerance in Spark:

- ▶ RDDs are immutable.
- ▶ We use higher-order functions like map, flatMap, filter to do *functional* transformations on this immutable data.
- ▶ A function for computing the dataset based on its parent RDDs also is part of an RDD's representation.



Lineages and Fault Tolerance

Lineages graphs are the key to fault tolerance in Spark.

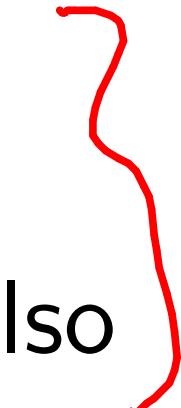
Ideas from **functional programming** enable fault tolerance in Spark:

- ▶ RDDs are immutable.
- ▶ We use higher-order functions like map, flatMap, filter to do *functional* transformations on this immutable data.
- ▶ A function for computing the dataset based on its parent RDDs also is part of an RDD's representation.

Along with keeping track of dependency information between partitions as well, this allows us to:

Recover from failures by recomputing lost partitions from lineage graphs.

Fault tolerance
w/out having
to checkpoint
+ write data
to disk!

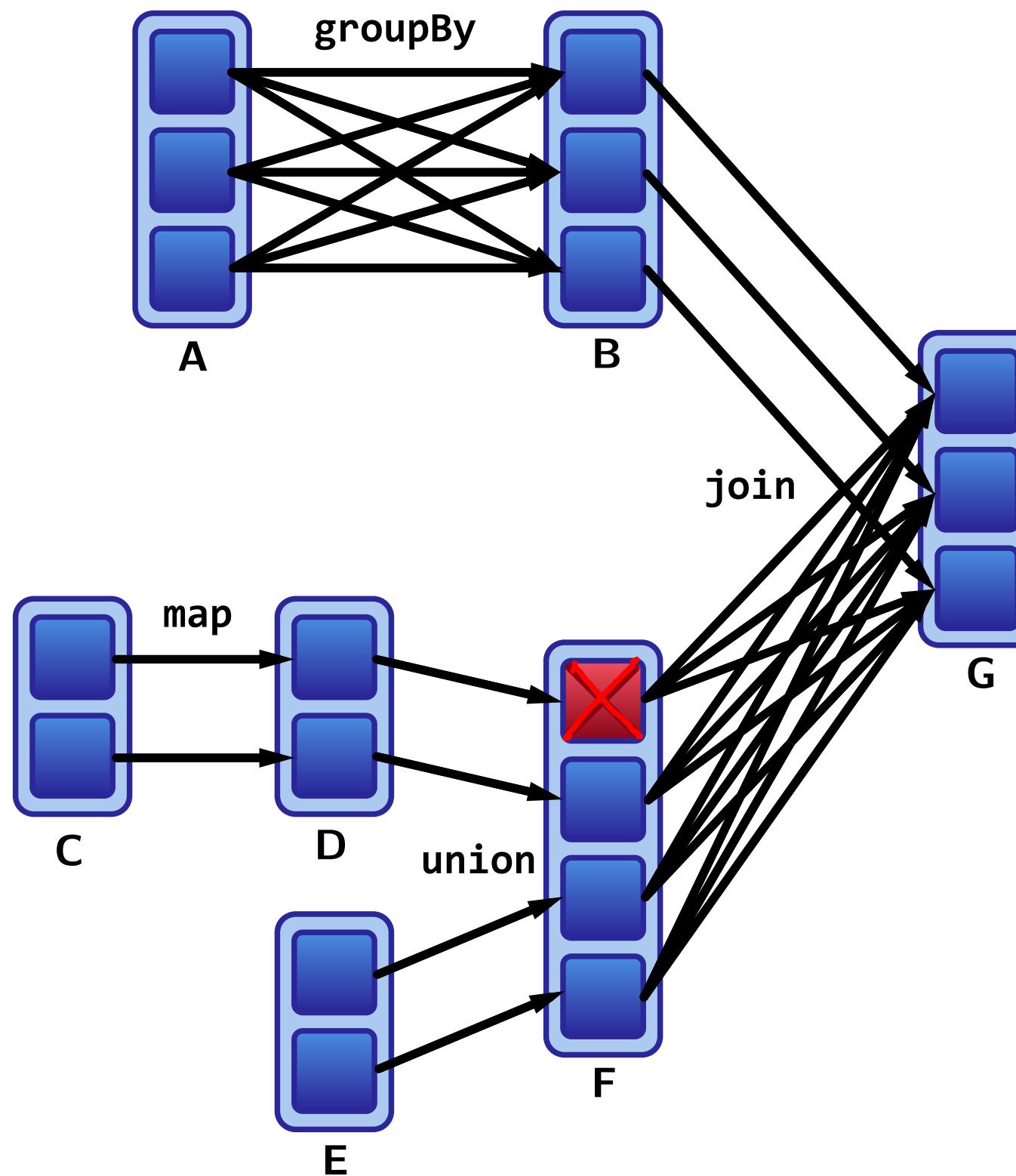


in-memory
+
fault tolerant!

Lineages and Fault Tolerance, Visually

Lineages graphs are the key to fault tolerance in Spark.

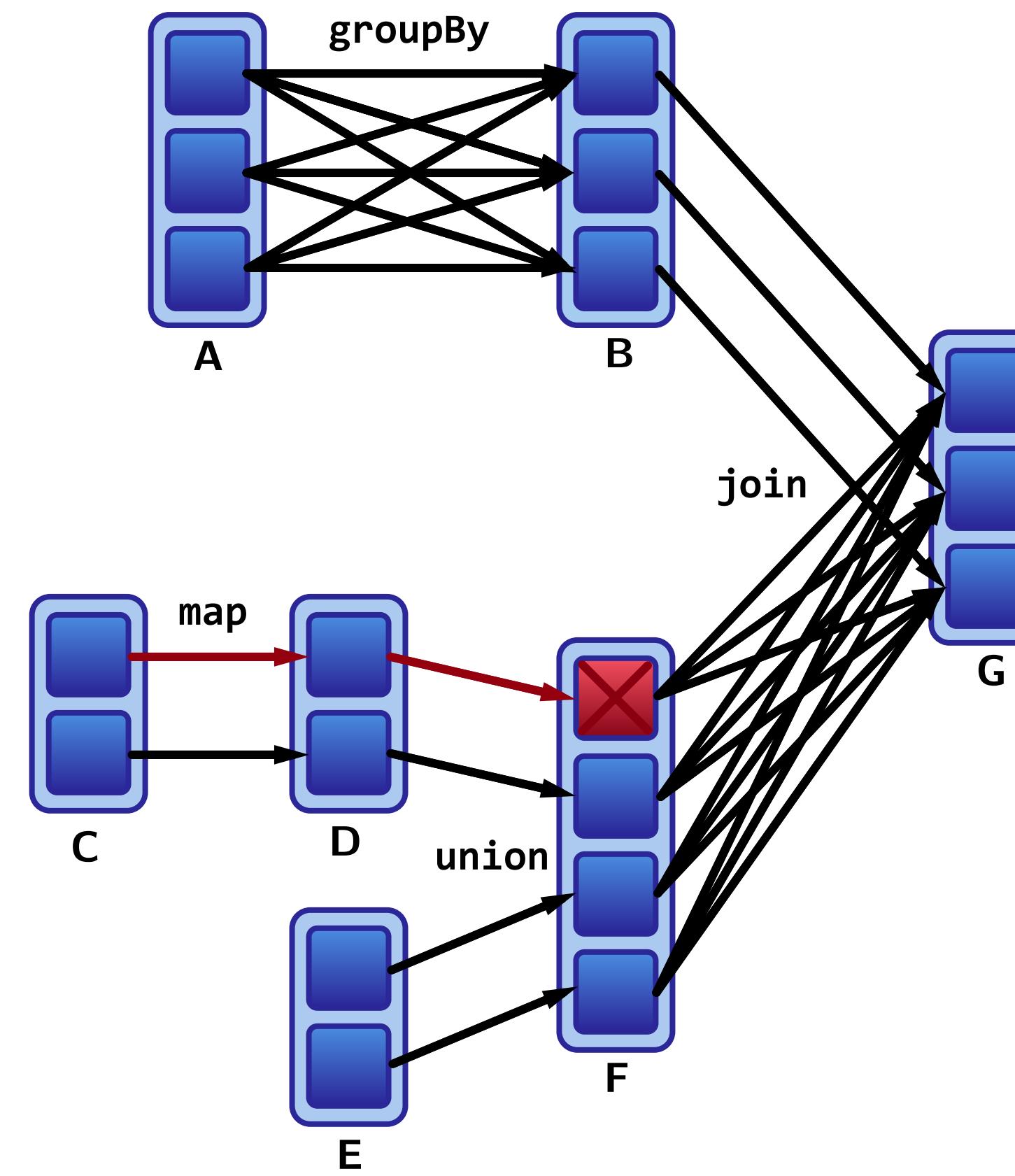
Let's assume one of our partitions from our previous example fails.



Lineages and Fault Tolerance, Visually

Lineages graphs are the key to fault tolerance in Spark.

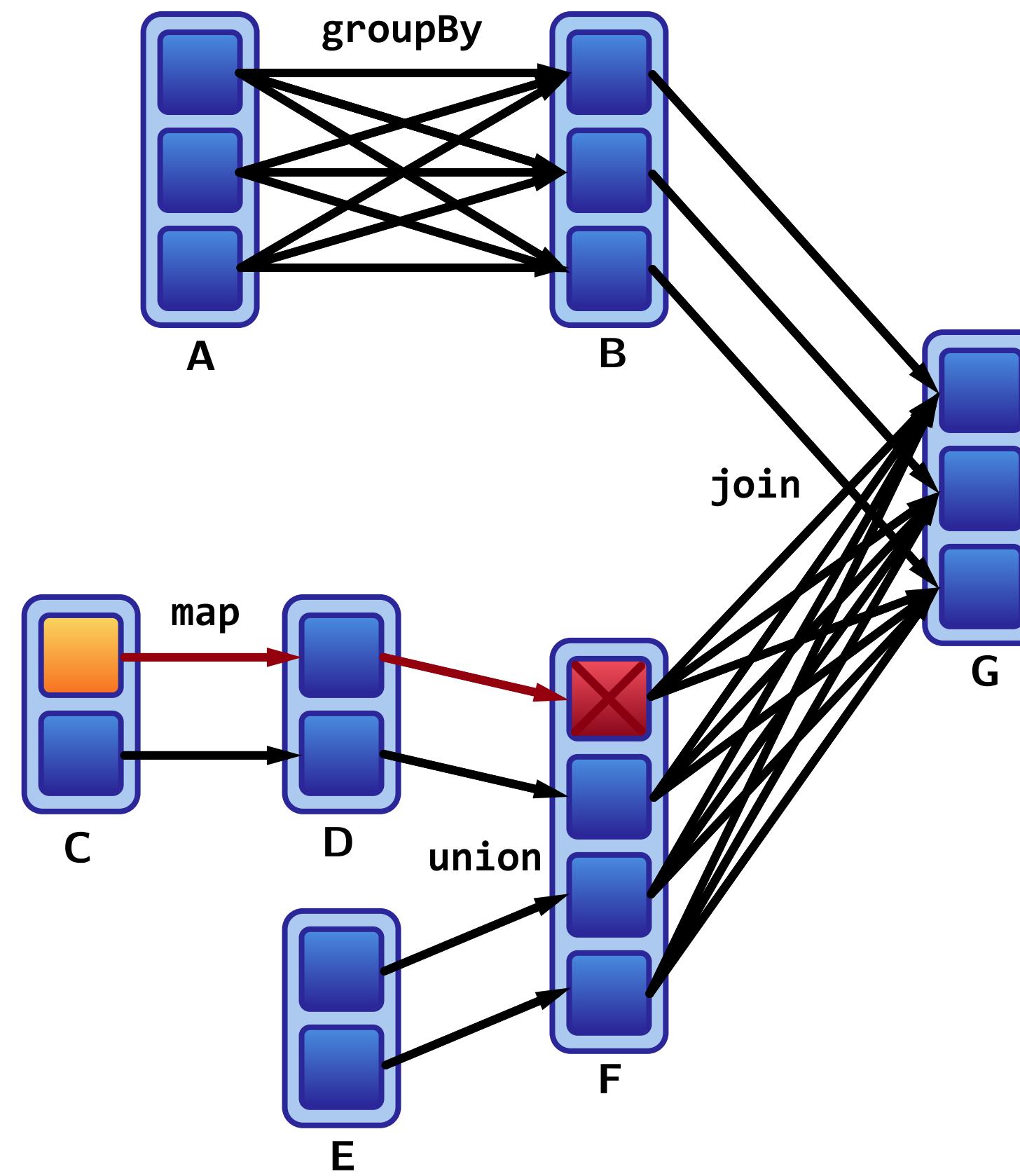
Let's assume one of our partitions from our previous example fails.



Lineages and Fault Tolerance, Visually

Lineages graphs are the key to fault tolerance in Spark.

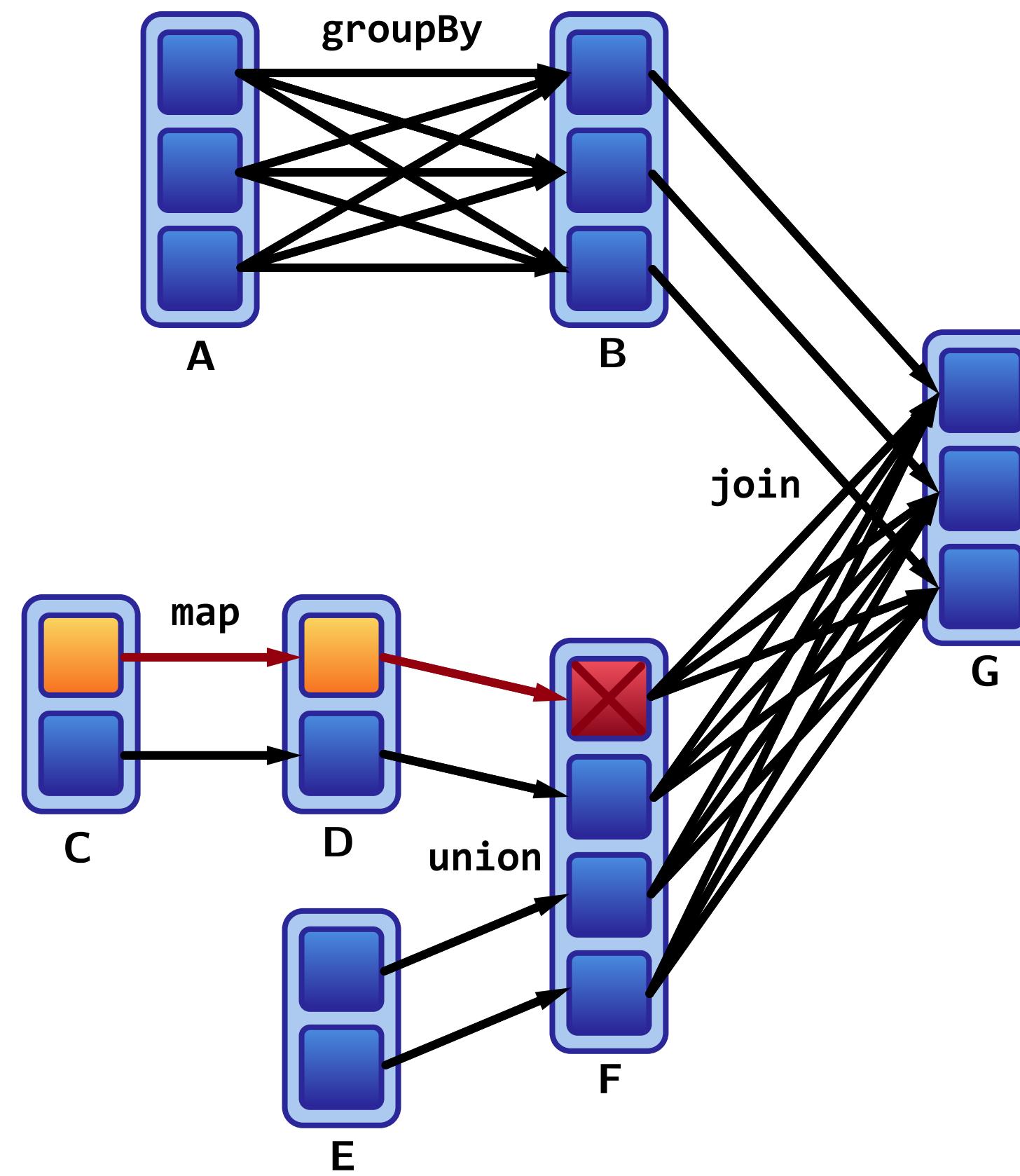
Let's assume one of our partitions from our previous example fails.



Lineages and Fault Tolerance, Visually

Lineages graphs are the key to fault tolerance in Spark.

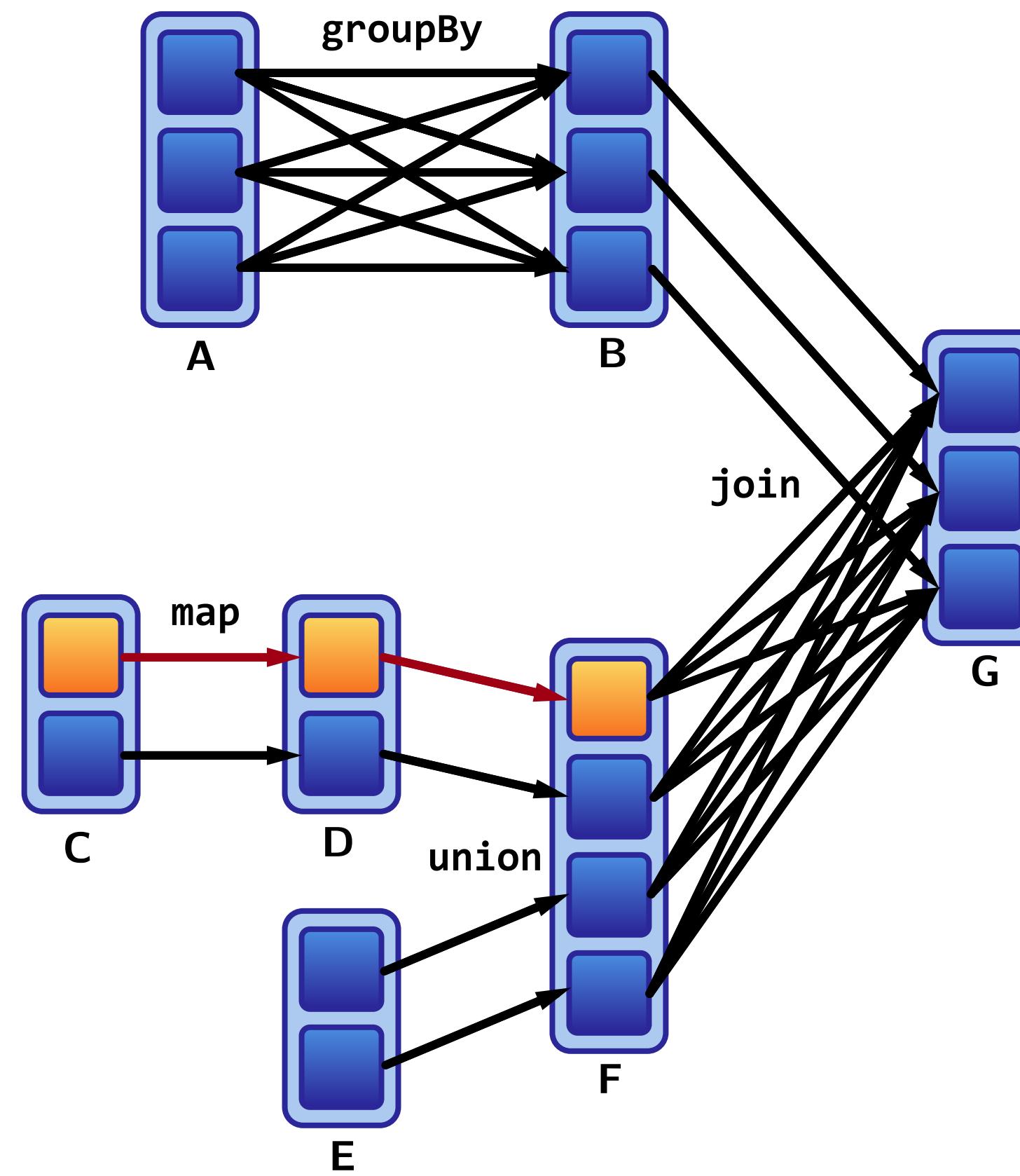
Let's assume one of our partitions from our previous example fails.



Lineages and Fault Tolerance, Visually

Lineages graphs are the key to fault tolerance in Spark.

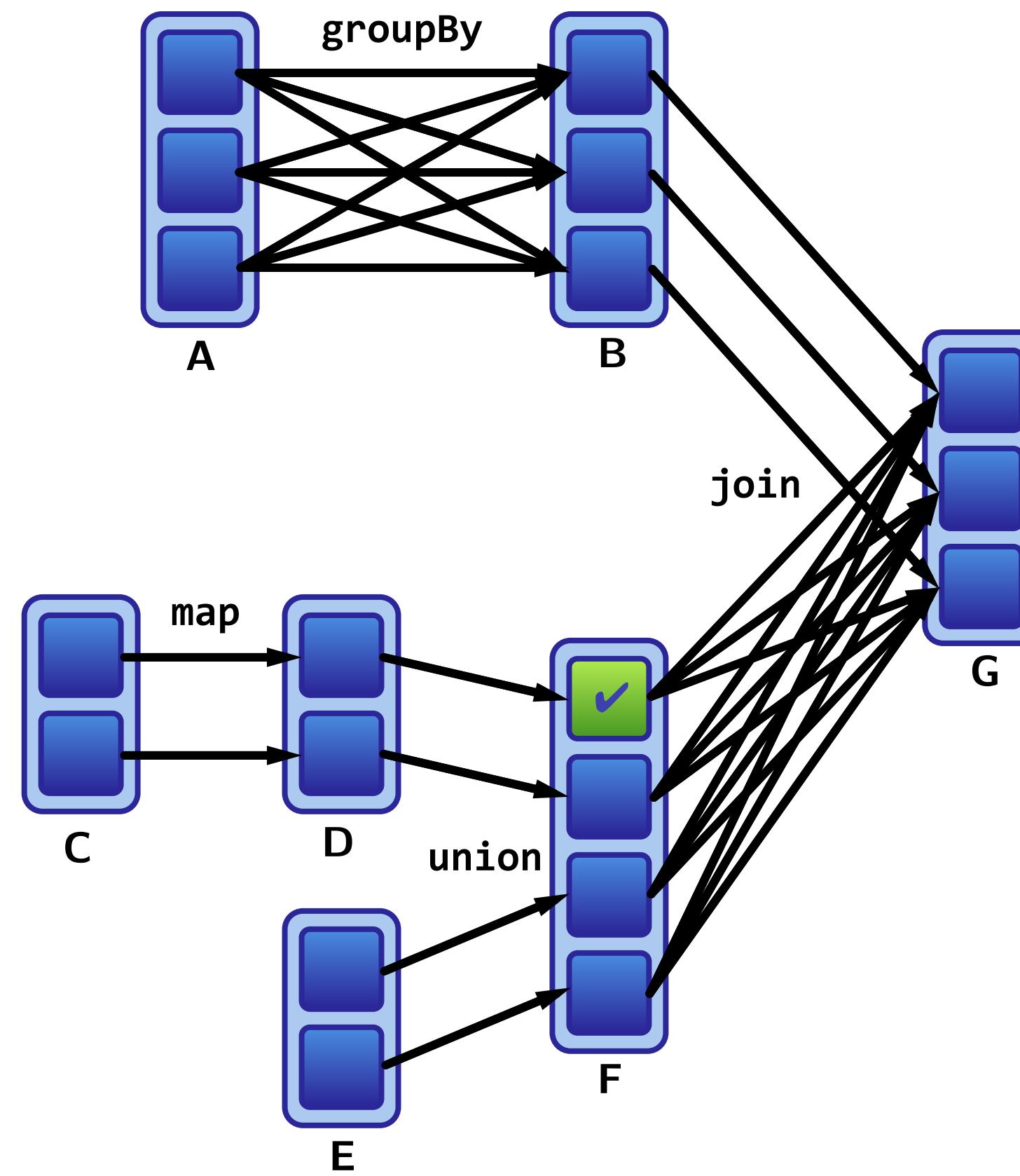
Let's assume one of our partitions from our previous example fails.



Lineages and Fault Tolerance, Visually

Lineages graphs are the key to fault tolerance in Spark.

Let's assume one of our partitions from our previous example fails.



Lineages and Fault Tolerance, Visually

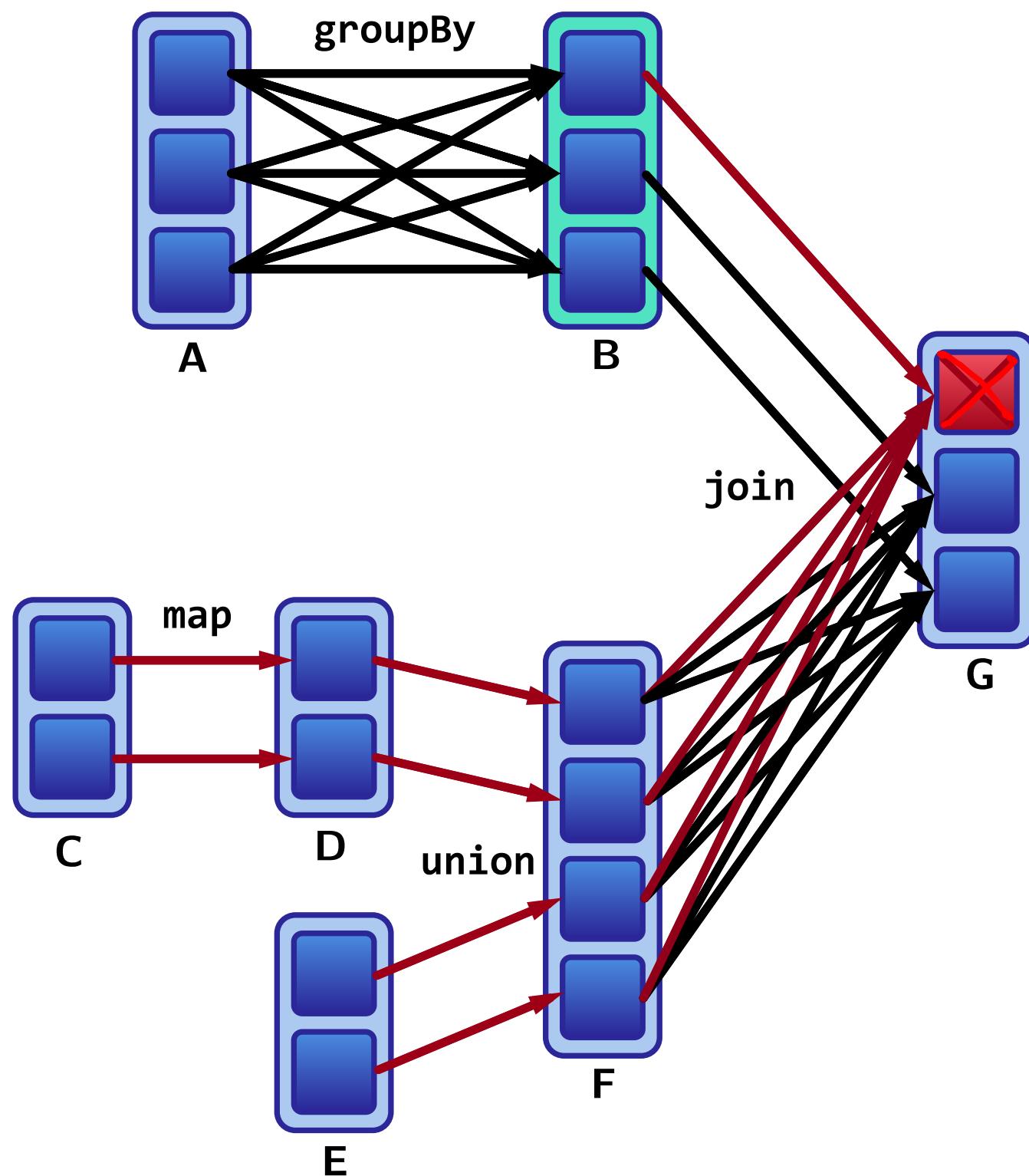
Lineages graphs are the key to fault tolerance in Spark.

Recomputing missing partitions fast for narrow dependencies. But slow for wide dependencies!

Lineages and Fault Tolerance, Visually

Lineages graphs are the key to fault tolerance in Spark.

Recomputing missing partitions fast for narrow dependencies. But ~~slow~~ for wide dependencies!





ÉCOLE POLYTECHNIQUE
FÉDÉRALE DE LAUSANNE

Structure and Optimization

Big Data Analysis with Scala and Spark

Heather Miller

Example: Selecting Scholarship Recipients

Let's imagine that we are an organization, CodeAward, offering scholarships to programmers who have overcome adversity. Let's say we have the following two datasets.

```
case class Demographic(id: Int,  
                      age: Int,  
                      codingBootcamp: Boolean,  
                      country: String,  
                      gender: String,  
                      isEthnicMinority: Boolean,  
                      servedInMilitary: Boolean)  
  
val demographics = sc.textfile(...)... // Pair RDD, (id, demographic)  
  
case class Finances(id: Int,  
                     hasDebt: Boolean,  
                     hasFinancialDependents: Boolean,  
                     hasStudentLoans: Boolean,  
                     income: Int)  
  
val finances = sc.textfile(...)... // Pair RDD, (id, finances)
```

Example: Selecting Scholarship Recipients

Our data sets include students from many countries, with many life and financial backgrounds. Now, let's imagine that our goal is to tally up and select students for a specific scholarship.

Example: Selecting Scholarship Recipients

Our data sets include students from many countries, with many life and financial backgrounds. Now, let's imagine that our goal is to tally up and select students for a specific scholarship.

As an example, Let's count:

- ▶ Swiss students
- ▶ who have debt & financial dependents

How might we implement this Spark program?

```
// Remember, RDDs available to us:  
val demographics = sc.textfile(...)... // Pair RDD, (id, demographic)  
val finances = sc.textfile(...)... // Pair RDD, (id, finances)
```

Example: Selecting Scholarship Recipients

Possibility 1:

$\uparrow id$
(Int, (Demographic, Finances))

```
demographics.join(finances)
    .filter { p =>
        p._2._1.country == "Switzerland" &&
        p._2._2.hasFinancialDependents &&
        p._2._2.hasDebt
    }.count
```

Example: Selecting Scholarship Recipients

Possibility 1:

```
demographics.join(finances)
    .filter { p =>
        p._2._1.country == "Switzerland" &&
        p._2._2.hasFinancialDependents &&
        p._2._2.hasDebt
    }.count
```

Steps:

1. Inner join first
2. Filter to select people in Switzerland
3. Filter to select people with debt & financial dependents

Example: Selecting Scholarship Recipients

Possibility 2:

```
val filtered
  = finances.filter(p => p._2.hasFinancialDependents && p._2.hasDebt)

demographics.filter(p => p._2.country == "Switzerland")
  .join(filtered)
  .count
```

Example: Selecting Scholarship Recipients

Possibility 2:

```
val filtered
  = finances.filter(p => p._2.hasFinancialDependents && p._2.hasDebt)

demographics.filter(p => p._2.country == "Switzerland")
  .join(filtered)
  .count
```

Steps:

1. Filter down the dataset first (look at only people with debt & financial dependents)
2. Filter to select people in Switzerland (look at only people in Switzerland)
3. Inner join on smaller, filtered down dataset

Example: Selecting Scholarship Recipients

Possibility 3:

```
val cartesian
  = demographics.cartesian(demographicsfinances)  
  
cartesian.filter {
  case (p1, p2) => p1._1 == p2._1      ← same ids
}  
.filter {
  case (p1, p2) => (p1._2.country == "Switzerland") &&
                    (p2._2.hasFinancialDependents) &&
                    (p2._2.hasDebt)
}.count
```

Example: Selecting Scholarship Recipients

Possibility 3:

```
val cartesian = demographics.cartesian(demographics)
```

```
cartesian.filter {
  case (p1, p2) => p1._1 == p2._1
}
.filter {
  case (p1, p2) => (p1._2.country == "Switzerland") &&
    (p2._2.hasFinancialDependents) &&
    (p2._2.hasDebt)
}.count
```

Steps:

1. Cartesian product on both datasets
2. Filter to select resulting of cartesian with same IDs
3. Filter to select people in Switzerland who have debt and financial dependents

Example: Selecting Scholarship Recipients

While for all three of these possible examples, the end result is the same, the time it takes to execute the job is vastly different.

Example: Selecting Scholarship Recipients

While for all three of these possible examples, the end result is the same, the time it takes to execute the job is vastly different.

150,000 people

Possibility 1

```
> ds.join(fs)
   .filter(p => p._2...
   .count
```

▶ (1) Spark Jobs

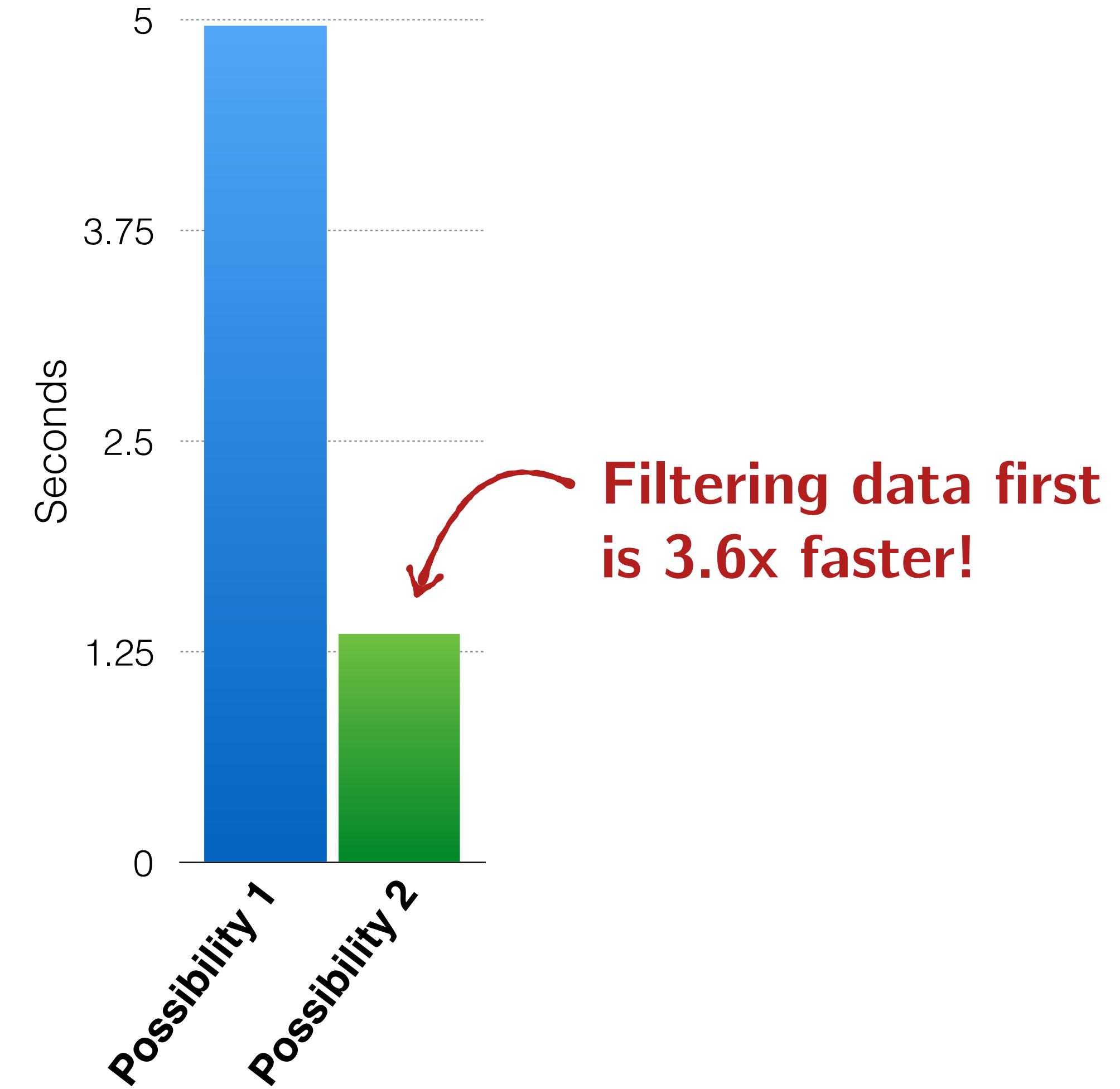
```
res0: Long = 10
Command took 4.97 seconds -
```

Possibility 2

```
> val fsi = fs.filter(
  ds.filter(p => p._2...
  .join(fsi)
  .count
```

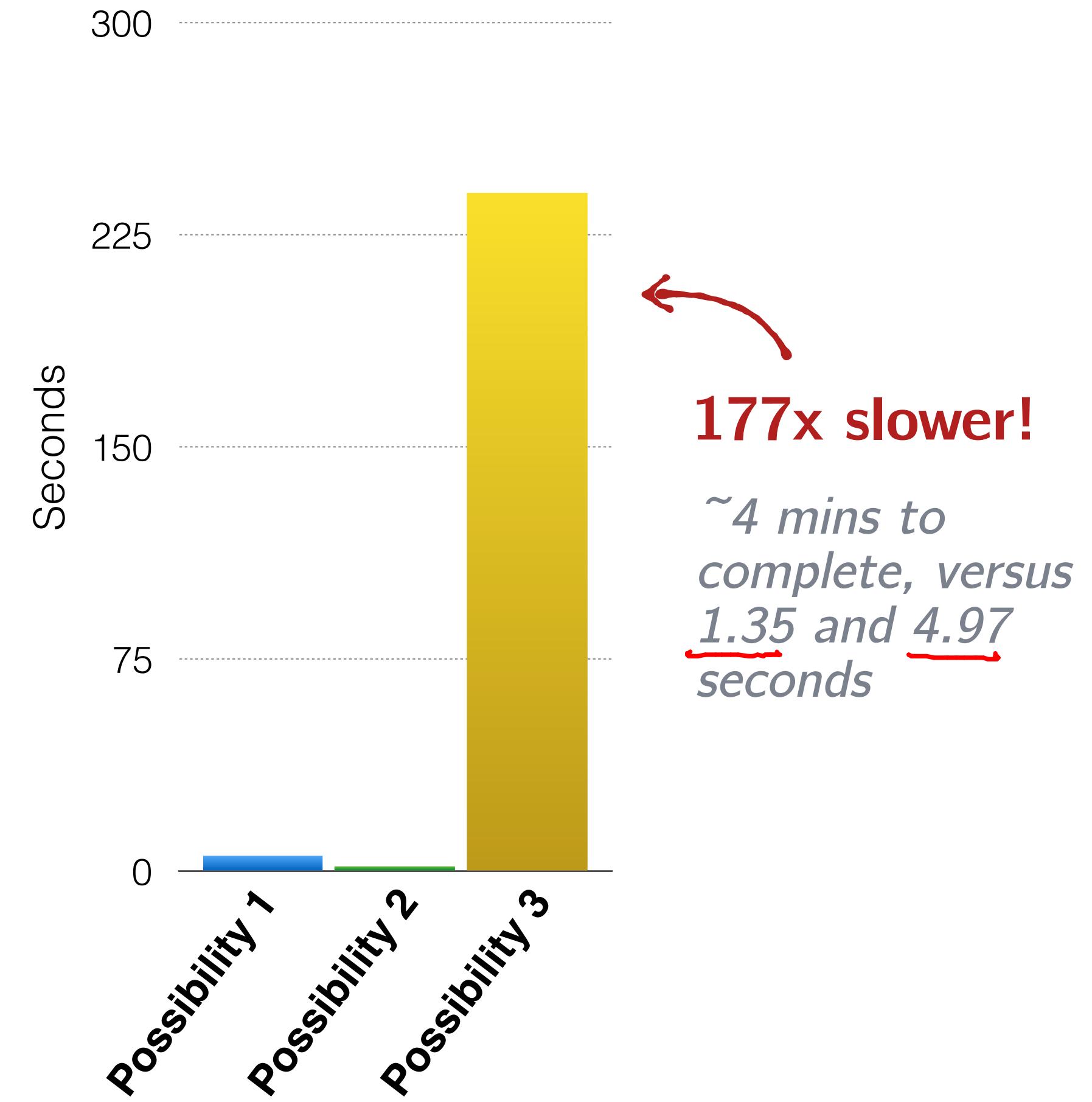
▶ (1) Spark Jobs

```
fsi: org.apache.spark.
res4: Long = 10
Command took 1.35 seconds -
```



Example: Selecting Scholarship Recipients

While for all three of these possible examples, the end result is the same, the time it takes to execute the job is vastly different.



Example: Selecting Scholarship Recipients

So far, a recurring theme has been that we have to think carefully about how our Spark jobs might actually be executed on the cluster in order to get good performance.

Example: Selecting Scholarship Recipients

So far, a recurring theme has been that we have to think carefully about how our Spark jobs might actually be executed on the cluster in order to get good performance.

Wouldn't it be nice if Spark automatically knew, if we wrote the code in possibility 3, that it could rewrite our code to possibility 2?

Example: Selecting Scholarship Recipients

So far, a recurring theme has been that we have to think carefully about how our Spark jobs might actually be executed on the cluster in order to get good performance.

Wouldn't it be nice if Spark automatically knew, if we wrote the code in possibility 3, that it could rewrite our code to possibility 2?

Given a bit of extra structural information, Spark can do many optimizations *for you!*

Structured vs Unstructured Data

All data isn't equal, structurally. It falls on a spectrum from unstructured to structured.

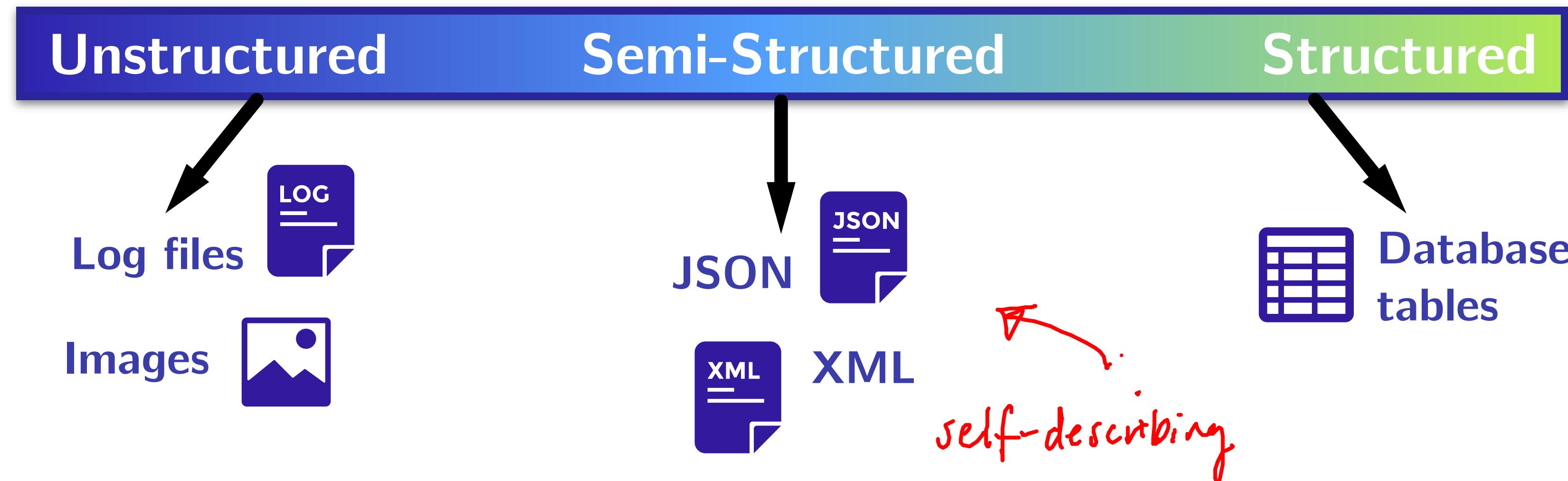
Unstructured

Semi-Structured

Structured

Structured vs Unstructured Data

All data isn't equal, structurally. It falls on a spectrum from unstructured to structured.



Structured Data vs RDDs

Spark + regular RDDs don't know anything about the *schema* of the data it's dealing with.

Structured Data vs RDDs

Spark + regular RDDs don't know anything about the *schema* of the data it's dealing with.

Given an arbitrary RDD, Spark knows that the RDD is parameterized with arbitrary types such as,

- ▶ Person
- ▶ Account
- ▶ Demographic



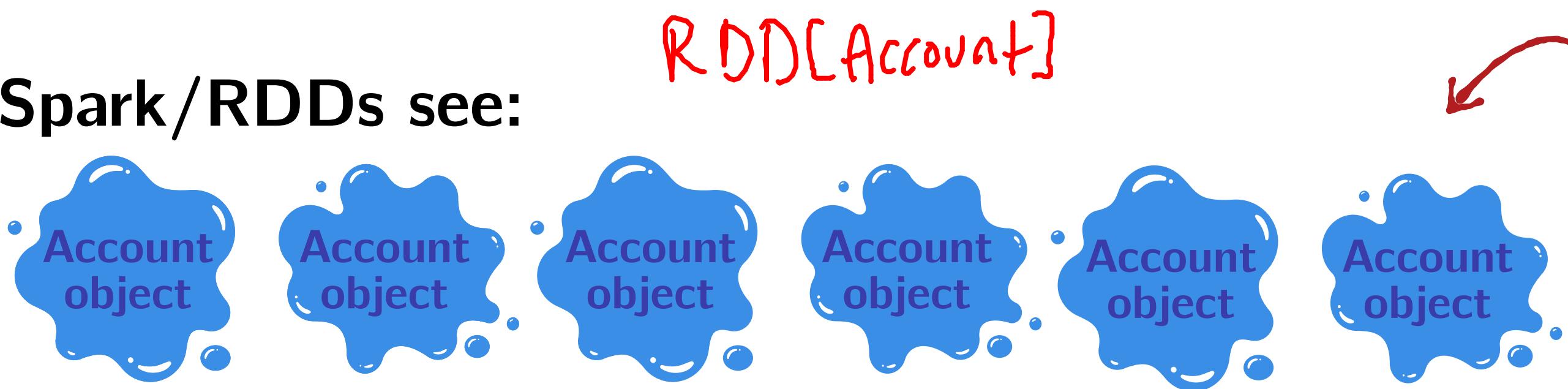
but it doesn't know anything about these types' structure.

Structured Data vs RDDs

Assuming we have a dataset of **Account** objects:

```
case class Account(name: String, balance: Double, risk: Boolean)
```

Spark/RDDs see:



Blobs of objects we know nothing about, except that they're called **Account**.

Spark can't see inside this object or analyze how it may be used, and to optimize based on that usage. It's opaque.

Structured Data vs RDDs

Assuming we have a dataset of **Account** objects:

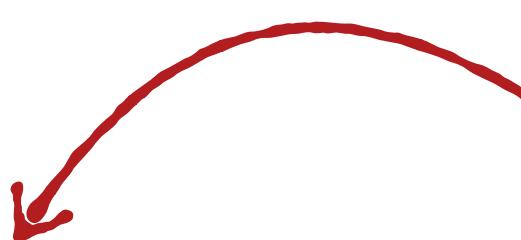
```
case class Account(name: String, balance: Double, risk: Boolean)
```

Spark/RDDs see:



A database/Hive sees:

name: String	balance: Double	risk: Boolean
name: String	balance: Double	risk: Boolean
name: String	balance: Double	risk: Boolean
name: String	balance: Double	risk: Boolean



Columns of named and typed values.

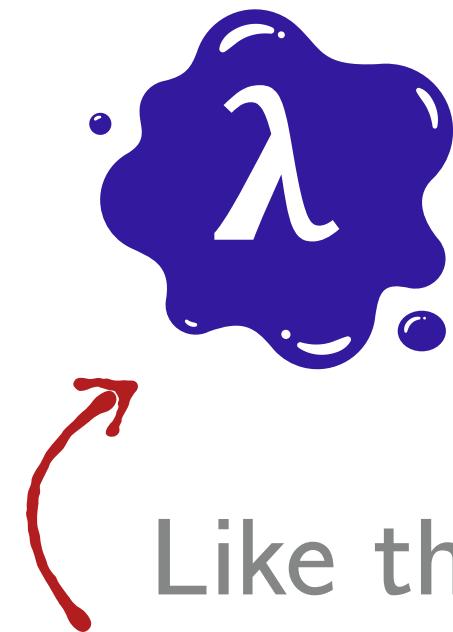
If Spark could see data this way, it could break up and only select the datatypes it needs to send around the cluster.

Structured vs Unstructured Computation

The same can be said about *computation*.

In Spark:

- ▶ We do **functional transformations** on data.
- ▶ We pass user-defined function literals to higher-order functions like `map`, `flatMap`, and `filter`.



Like the data Spark operates on, function literals too are completely opaque to Spark.

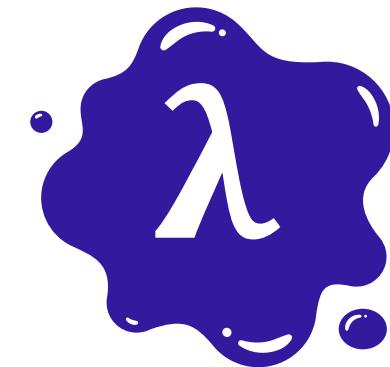
A user can do anything inside of one of these, and all Spark can see is something like:
`$anon$1@604f1a67`

Structured vs Unstructured Computation

The same can be said about *computation*.

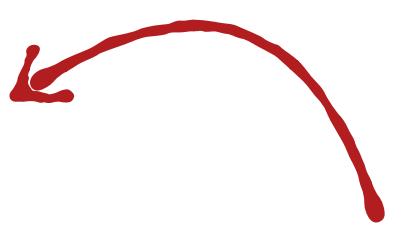
In Spark:

- ▶ We do **functional transformations** on data.
- ▶ We pass user-defined function literals to higher-order functions like `map`, `flatMap`, and `filter`.



In a database/Hive:

- ▶ We do **declarative transformations** on data.
- ▶ Specialized/structured, pre-defined operations.



Fixed set of operations,
fixed set of types they
operate on.

Optimizations the norm!

Structured vs Unstructured

In summary:

Spark RDDs: *as we know them so far*



Databases/Hive:

name: String	balance: Double	risk: Boolean
name: String	balance: Double	risk: Boolean
name: String	balance: Double	risk: Boolean
name: String	balance: Double	risk: Boolean

**SELECT
WHERE
ORDER BY
GROUP BY
COUNT**

Structured vs Unstructured

In summary:

Spark RDDs: *as we know them so far*



**Not much structure.
Difficult to
aggressively optimize.**

Databases/Hive:

name: String	balance: Double	risk: Boolean
name: String	balance: Double	risk: Boolean
name: String	balance: Double	risk: Boolean
name: String	balance: Double	risk: Boolean

**SELECT
WHERE
ORDER BY
GROUP BY
COUNT**

Structured vs Unstructured

In summary:

Spark RDDs: *as we know them so far*



**Not much structure.
Difficult to aggressively optimize.**

Databases/Hive:

name: String	balance: Double	risk: Boolean
name: String	balance: Double	risk: Boolean
name: String	balance: Double	risk: Boolean
name: String	balance: Double	risk: Boolean

**SELECT
WHERE
ORDER BY
GROUP BY
COUNT**

**Lots of structure.
Lots of optimization opportunities!**

Optimizations + Spark?

RDDs operate on unstructured data, and there are few limits on computation; your computations are defined as functions that you've written yourself, on your own data types.

But as we saw, we have to do all the optimization work ourselves!

Optimizations + Spark?

RDDs operate on unstructured data, and there are few limits on computation; your computations are defined as functions that you've written yourself, on your own data types.

But as we saw, we have to do all the optimization work ourselves!

Wouldn't it be nice if Spark could do some of these optimizations for us?

Optimizations + Spark?

RDDs operate on unstructured data, and there are few limits on computation; your computations are defined as functions that you've written yourself, on your own data types.

But as we saw, we have to do all the optimization work ourselves!

Wouldn't it be nice if Spark could do some of these optimizations for us?

Spark SQL makes this possible!

We've got to give up some of the freedom, flexibility, and generality of the functional collections API in order to give Spark more opportunities to optimize though.



ÉCOLE POLYTECHNIQUE
FÉDÉRALE DE LAUSANNE

Spark SQL

Big Data Analysis with Scala and Spark

Heather Miller

Relational Databases

SQL is the lingua franca for doing analytics.

Relational Databases

SQL is the lingua franca for doing analytics.

But it's a pain in the neck to connect big data processing pipelines like Spark or Hadoop to an SQL database.

Wouldn't it be nice...

- ▶ if it were possible to seamlessly intermix SQL queries with Scala?
- ▶ to get all of the optimizations we're used to in the databases community on Spark jobs?

Relational Databases

SQL is the lingua franca for doing analytics.

But it's a pain in the neck to connect big data processing pipelines like Spark or Hadoop to an SQL database.

Wouldn't it be nice...

- ▶ if it were possible to seamlessly intermix SQL queries with Scala?
- ▶ to get all of the optimizations we're used to in the databases community on Spark jobs?

Spark SQL delivers both!

Spark SQL: Goals

Three main goals:

1. Support **relational processing** both within Spark programs (on RDDs) and on external data sources with a friendly API.

Sometimes it's more desirable to express a computation in SQL syntax than with functional APIs and vice versa.

Spark SQL: Goals

Three main goals:

1. Support **relational processing** both within Spark programs (on RDDs) and on external data sources with a friendly API.
2. High performance, achieved by using techniques from research in databases.
3. Easily support new data sources such as semi-structured data and external databases.

Spark SQL

Spark SQL is a component of the Spark stack.

- ▶ It is a Spark module for structured data processing.
- ▶ It is implemented as a library on top of Spark.

Spark SQL

Spark SQL is a component of the Spark stack.

- ▶ It is a Spark module for structured data processing.
- ▶ It is implemented as a library on top of Spark.

Three main APIs:

- ▶ **SQL literal syntax**
- ▶ **DataFrames**
- ▶ **Datasets**

Spark SQL

Spark SQL is a component of the Spark stack.

- ▶ It is a Spark module for structured data processing.
- ▶ It is implemented as a library on top of Spark.

Three main APIs:

- ▶ **SQL literal syntax**
- ▶ **DataFrames**
- ▶ **Datasets**

Two specialized backend components:

- ▶ **Catalyst**, query optimizer.
- ▶ **Tungsten**, off-heap serializer.

Spark SQL

Spark SQL is a component of the Spark stack.

- ▶ It is a Spark module for structured data processing.
- ▶ It is implemented as a library on top of Spark.

Visually, Spark SQL relates to the rest of Spark like this:

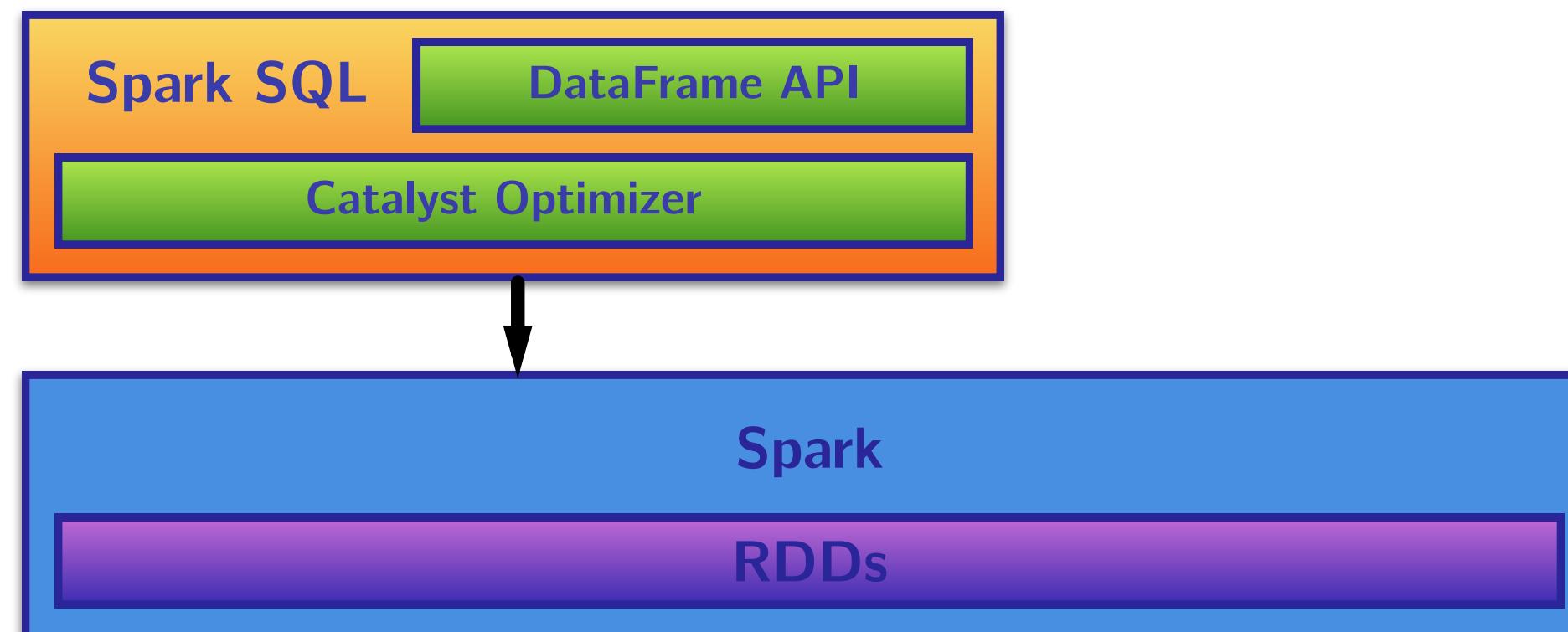


Spark SQL

Spark SQL is a component of the Spark stack.

- ▶ It is a Spark module for structured data processing.
- ▶ It is implemented as a library on top of Spark.

Visually, Spark SQL relates to the rest of Spark like this:

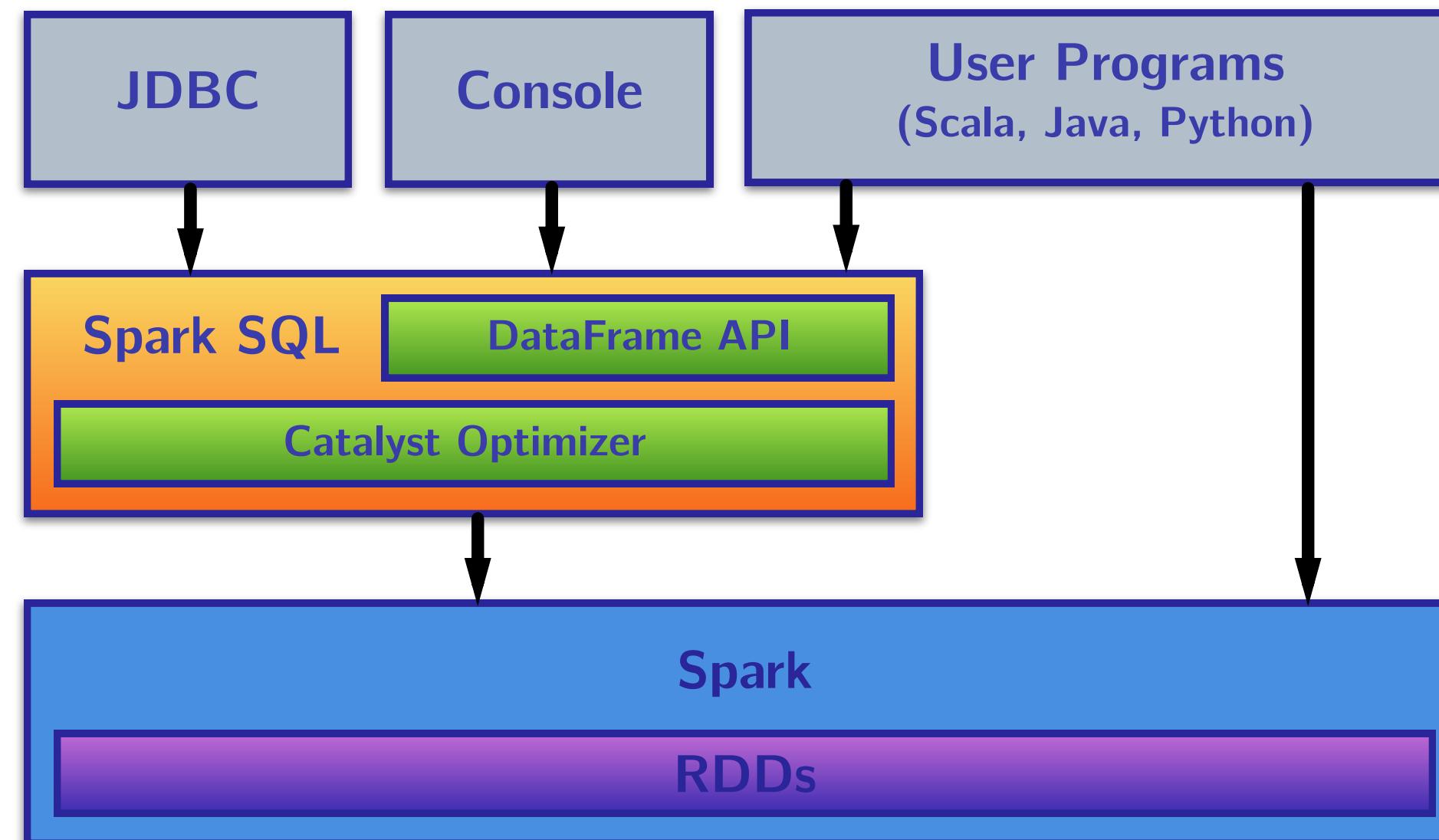


Spark SQL

Spark SQL is a component of the Spark stack.

- ▶ It is a Spark module for structured data processing.
- ▶ It is implemented as a library on top of Spark.

Visually, Spark SQL relates to the rest of Spark like this:



Relational Queries (SQL)

Everything about SQL is structured.

In fact, SQL stands for *structural query language*.

- ▶ There are a set of fixed data types. Int, Long, String, etc.
- ▶ There are fixed set of operations. SELECT, WHERE, GROUP BY, etc.

Research and industry surrounding relational databases has focused on exploiting this rigidness to get all kinds of performance speedups.

Relational Queries (SQL)

Everything about SQL is structured.

In fact, SQL stands for *structural query language*.

- ▶ There are a set of fixed data types. Int, Long, String, etc.
- ▶ There are fixed set of operations. SELECT, WHERE, GROUP BY, etc.

Research and industry surrounding relational databases has focused on exploiting this rigidness to get all kinds of performance speedups.

Let's quickly establish a common set of vocabulary and a baseline understanding of SQL.

Relational Queries (SQL)

Data organized into one or more **tables**

Customer _ Name	Destination	Ticket _ Price
"Weitz"	"Luzern"	53.20
"Schinz"	"Zürich"	32.40
"Dubois"	"Neuchâtel"	12.50
"Hug"	"Basel"	32.10
"Strub"	"Winterthur"	9.60
"Chapuis"	"Lausanne"	6.60
"Smith"	"Genève"	12.70
"Weitz"	"Bern"	21.40

Relational Queries (SQL)

Data organized into one or more **tables**

- ▶ Tables contain **columns** and **rows**.

columns

Customer_Name	Destination	Ticket_Price
"Weitz"	"Luzern"	53.20
"Schinz"	"Zürich"	32.40
"Dubois"	"Neuchâtel"	12.50
"Hug"	"Basel"	32.10
"Strub"	"Winterthur"	9.60
"Chapuis"	"Lausanne"	6.60
"Smith"	"Genève"	12.70
"Weitz"	"Bern"	21.40

Relational Queries (SQL)

Data organized into one or more **tables**

- ▶ Tables contain *columns* and **rows**.

rows

Customer_Name	Destination	Ticket_Price
"Weitz"	"Luzern"	53.20
"Schinz"	"Zürich"	32.40
"Dubois"	"Neuchâtel"	12.50
"Hug"	"Basel"	32.10
"Strub"	"Winterthur"	9.60
"Chapuis"	"Lausanne"	6.60
"Smith"	"Genève"	12.70
"Weitz"	"Bern"	21.40

Relational Queries (SQL)

Data organized into one or more **tables**

- ▶ Tables contain **columns** and **rows**.
- ▶ Tables typically represent a collection of objects of a certain type, such as **customers** or **products**

SBB customers dataset

Customer_Name	Destination	Ticket_Price
"Weitz"	"Luzern"	53.20
"Schinz"	"Zürich"	32.40
"Dubois"	"Neuchâtel"	12.50
"Hug"	"Basel"	32.10
"Strub"	"Winterthur"	9.60
"Chapuis"	"Lausanne"	6.60
"Smith"	"Genève"	12.70
"Weitz"	"Bern"	21.40

Relational Queries (SQL)

Data organized into one or more **tables**

- ▶ Tables contain **columns** and **rows**.
- ▶ Tables typically represent a collection of objects of a certain type, such as **customers** or **products**

A **relation** is just a table.

Attributes are columns.

 **attribute**

Customer_Name	Destination	Ticket_Price
"Weitz"	"Luzern"	53.20
"Schinz"	"Zürich"	32.40
"Dubois"	"Neuchâtel"	12.50
"Hug"	"Basel"	32.10
"Strub"	"Winterthur"	9.60
"Chapuis"	"Lausanne"	6.60
"Smith"	"Genève"	12.70
"Weitz"	"Bern"	21.40

Relational Queries (SQL)

Data organized into one or more **tables**

- ▶ Tables contain **columns** and **rows**.
- ▶ Tables typically represent a collection of objects of a certain type, such as **customers** or **products**

A *relation* is just a table.

Attributes are columns.

Rows are *records* or *tuples*

record
/tuple

Customer_Name	Destination	Ticket_Price
"Weitz"	"Luzern"	53.20
"Schinz"	"Zürich"	32.40
"Dubois"	"Neuchâtel"	12.50
"Hug"	"Basel"	32.10
"Strub"	"Winterthur"	9.60
"Chapuis"	"Lausanne"	6.60
"Smith"	"Genève"	12.70
"Weitz"	"Bern"	21.40

Spark SQL

DataFrame is Spark SQL's core abstraction.

Conceptually equivalent to a table in a relational database.

DataFrame, is a table, sort of.

Customer_Name	Destination	Ticket_Price
"Weitz"	"Luzern"	53.20
"Schinz"	"Zürich"	32.40
"Dubois"	"Neuchâtel"	12.50
"Hug"	"Basel"	32.10
"Strub"	"Winterthur"	9.60
"Chapuis"	"Lausanne"	6.60
"Smith"	"Genève"	12.70
"Weitz"	"Bern"	21.40

Spark SQL

DataFrame is Spark SQL's core abstraction.

Conceptually equivalent to a table in a relational database.

DataFrames are, *conceptually*, RDDs full of records with a known schema

distributed collection of rows/records.

DataFrame, is a table, sort of.

Customer_Name	Destination	Ticket_Price
"Weitz"	"Luzern"	53.20
"Schinz"	"Zürich"	32.40
"Dubois"	"Neuchâtel"	12.50
"Hug"	"Basel"	32.10
"Strub"	"Winterthur"	9.60
"Chapuis"	"Lausanne"	6.60
"Smith"	"Genève"	12.70
"Weitz"	"Bern"	21.40

Spark SQL

DataFrame is Spark SQL's core abstraction.

Conceptually equivalent to a table in a relational database.

DataFrames are, *conceptually*, RDDs full of records **with a known schema**

Unlike RDDs though, DataFrames require some kind of schema info!

DataFrame, is a table, sort of.

Customer_Name	Destination	Ticket_Price
"Weitz"	"Luzern"	53.20
"Schinz"	"Zürich"	32.40
"Dubois"	"Neuchâtel"	12.50
"Hug"	"Basel"	32.10
"Strub"	"Winterthur"	9.60
"Chapuis"	"Lausanne"	6.60
"Smith"	"Genève"	12.70
"Weitz"	"Bern"	21.40

Spark SQL

DataFrame is Spark SQL's core abstraction.

Conceptually equivalent to a table in a relational database.

DataFrames are, *conceptually*, RDDs full of records with a known schema

!! !!

DataFrames are **untyped!**

That is, the Scala compiler doesn't check the types in its schema!

DataFrames contain Rows which can contain any schema.

DataFrame, is a table, sort of.

Customer_Name	Destination	Ticket_Price
"Weitz"	"Luzern"	53.20
"Schinz"	"Zürich"	32.40
"Dubois"	"Neuchâtel"	12.50
"Hug"	"Basel"	32.10
"Strub"	"Winterthur"	9.60
"Chapuis"	"Lausanne"	6.60
"Smith"	"Genève"	12.70
"Weitz"	"Bern"	21.40

RDD[T]
DataFrame

Spark SQL

DataFrame is Spark SQL's core abstraction.

Conceptually equivalent to a table in a relational database.

DataFrames are, *conceptually*, RDDs full of records with a known schema

DataFrames are **untypesd**!

That is, the Scala compiler doesn't check the types in its schema!

Transformations on DataFrames are also known as **untypesd transformations**

DataFrame, is a table, sort of.

Customer_Name	Destination	Ticket_Price
"Weitz"	"Luzern"	53.20
"Schinz"	"Zürich"	32.40
"Dubois"	"Neuchâtel"	12.50
"Hug"	"Basel"	32.10
"Strub"	"Winterthur"	9.60
"Chapuis"	"Lausanne"	6.60
"Smith"	"Genève"	12.70
"Weitz"	"Bern"	21.40

SparkSession

To get started using Spark SQL, everything starts with the `SparkSession`

SparkSession

To get started using Spark SQL, everything starts with the `SparkSession`

```
import org.apache.spark.sql.SparkSession

val spark = SparkSession
  .builder()
  .appName("My App")
  //.config("spark.some.config.option", "some-value")
  .getOrCreate()
```

Creating DataFrames

DataFrames can be created in two ways:

1. From an existing RDD.

Either with schema inference, or with an explicit schema.

2. Reading in a specific **data source** from file.

Common structured or semi-structured formats such as JSON.

Creating DataFrames

(1a) Create DataFrame from RDD, schema reflectively inferred

Given pair RDD, RDD[(T1, T2, ... TN)], a DataFrame can be created with its schema automatically inferred by simply using the toDF method.

```
val tupleRDD = ... // Assume RDD[(Int, String String, String)]  
val tupleDF = tupleRDD.toDF("id", "name", "city", "country") // column names
```

Note: if you use toDF without arguments, Spark will assign numbers as attributes (column names) to your DataFrame.

-1 -2 -3

Creating DataFrames

(1a) Create DataFrame from RDD, schema reflectively inferred

Given pair RDD, RDD[(T1, T2, ... TN)], a DataFrame can be created with its schema automatically inferred by simply using the toDF method.

```
val tupleRDD = ... // Assume RDD[(Int, String, String, String)]  
val tupleDF = tupleRDD.toDF("id", "name", "city", "country") // column names
```

Note: if you use toDF without arguments, Spark will assign numbers as attributes (column names) to your DataFrame.

If you already have an RDD containing some kind of case class instance, then Spark can infer the attributes from the case class's fields.

```
case class Person(id: Int, name: String, city: String)  
val peopleRDD = ... // Assume RDD[Person]  
val peopleDF = peopleRDD.toDF
```



Creating DataFrames

(1b) Create DataFrame from existing RDD, schema explicitly specified

Sometimes it's not possible to create a DataFrame with a pre-determined case class as its schema. For these cases, it's possible to explicitly specify a schema.

It takes three steps:

- ▶ Create an RDD of Rows from the original RDD.
- ▶ Create the schema represented by a StructType matching the structure of Rows in the RDD created in Step 1.
- ▶ Apply the schema to the RDD of Rows via createDataFrame method provided by SparkSession.

Given:

```
case class Person(name: String, age: Int)  
val peopleRdd = sc.textFile(...) // Assume RDD[Person]
```

Creating DataFrames

(1b) Create DataFrame from existing RDD, schema explicitly specified

```
// The schema is encoded in a string
val schemaString = "name age"

// Generate the schema based on the string of schema
val fields = schemaString.split(" ")
    .map(fieldName => StructField(fieldName, StringType, nullable = true))
val schema = StructType(fields)

// Convert records of the RDD (people) to Rows
val rowRDD = peopleRDD
    .map(_.split(","))
    .map(attributes => Row(attributes(0), attributes(1).trim))

// Apply the schema to the RDD
val peopleDF = spark.createDataFrame(rowRDD, schema)
```

Creating DataFrames

(2) Create DataFrame by reading in a data source from file.

Using the SparkSession object, you can read in semi-structured/structured data by using the read method. For example, to read in data and infer a schema from a JSON file:

```
// 'spark' is the SparkSession object we created a few slides back  
val df = spark.read.json("examples/src/main/resources/people.json")
```

Creating DataFrames

(2) Create DataFrame by reading in a data source from file.

Using the SparkSession object, you can read in semi-structured/structured data by using the read method. For example, to read in data and infer a schema from a JSON file:

```
// 'spark' is the SparkSession object we created a few slides back  
val df = spark.read.json("examples/src/main/resources/people.json")
```

Semi-structured/Structured data sources Spark SQL can directly create DataFrames from:

- ▶ JSON
- ▶ CSV
- ▶ Parquet
- ▶ JDBC

*To see a list of all available methods for directly reading in semi-structured/structured data, see the latest API docs for DataFrameReader:
<http://spark.apache.org/docs/latest/api/scala/index.html#org.apache.spark.sql.DataFrameReader>*

SQL Literals

Once you have a DataFrame to operate on, you can now freely write familiar SQL syntax to operate on your dataset!

SQL Literals

Once you have a DataFrame to operate on, you can now freely write familiar SQL syntax to operate on your dataset!

Given:

A DataFrame called peopleDF, we just have to register our DataFrame as a temporary SQL view first:

```
// Register the DataFrame as a SQL temporary view  
peopleDF.createOrReplaceTempView("people")  
// This essentially gives a name to our DataFrame in SQL  
// so we can refer to it in an SQL FROM statement
```

SQL Literals

Once you have a DataFrame to operate on, you can now freely write familiar SQL syntax to operate on your dataset!

Given:

A DataFrame called peopleDF, we just have to register our DataFrame as a temporary SQL view first:

```
// Register the DataFrame as a SQL temporary view
peopleDF.createOrReplaceTempView("people")
// This essentially gives a name to our DataFrame in SQL
// so we can refer to it in an SQL FROM statement

// SQL literals can be passed to Spark SQL's sql method
val adultsDF
  = spark.sql("SELECT * FROM people WHERE age > 17")
```

SQL Literals

The SQL statements available to you are largely what's available in HiveQL. This includes standard SQL statements such as:

SQL Literals

The SQL statements available to you are largely what's available in HiveQL. This includes standard SQL statements such as:

- ▶ SELECT
- ▶ FROM
- ▶ WHERE
- ▶ COUNT
- ▶ HAVING
- ▶ GROUP BY
- ▶ ORDER BY
- ▶ SORT BY
- ▶ DISTINCT
- ▶ JOIN
- ▶ (LEFT|RIGHT|FULL) OUTER JOIN
- ▶ Subqueries: `SELECT col FROM (SELECT a + b AS col from t1) t2`



Supported Spark SQL syntax:

https://docs.datastax.com/en/datastax_enterprise/4.6/datastax_enterprise/spark/sparkSqlSupportedSyntax.html

For a HiveQL cheatsheet:

<https://hortonworks.com/blog/hive-cheat-sheet-for-sql-users/>

For an updated list of supported Hive features in Spark SQL, the official Spark SQL docs enumerate:

<https://spark.apache.org/docs/latest/sql-programming-guide.html#supported-hive-features>

A More Interesting SQL Query

Let's assume we have a DataFrame representing a data set of employees:

```
case class Employee(id: Int, fname: String, lname: String, age: Int, city: String)

// DataFrame with schema defined in Employee case class
val employeeDF = sc.parallelize(...).toDF
```

A More Interesting SQL Query

Let's assume we have a DataFrame representing a data set of employees:

```
case class Employee(id: Int, fname: String, lname: String, age: Int, city: String)  
  
// DataFrame with schema defined in Employee case class  
val employeeDF = sc.parallelize(...).toDF
```

Let's query this data set to obtain just the IDs and last names of employees working in a specific city, say, Sydney, Australia. Let's sort our result in order of increasing employee ID.

What would this SQL query look like?

A More Interesting SQL Query

Let's assume we have a DataFrame representing a data set of employees:

```
case class Employee(id: Int, fname: String, lname: String, age: Int, city: String)  
  
// DataFrame with schema defined in Employee case class  
val employeeDF = sc.parallelize(...).toDF  
  
val sydneyEmployeesDF  
= spark.sql("""SELECT id, lname  
          FROM employees  
         WHERE city = "Sydney"  
        ORDER BY id""")  
  
registered "employees"
```

A More Interesting SQL Query

Let's assume we have a DataFrame representing a data set of employees:

```
case class Employee(id: Int, fname: String, lname: String, age: Int, city: String)

// DataFrame with schema defined in Employee case class
val employeeDF = sc.parallelize(...).toDF

val sydneyEmployeesDF
  = spark.sql("""SELECT id, lname
                FROM employees
               WHERE city = "Sydney"
              ORDER BY id""")
```

Pretty simple.

A More Interesting SQL Query

Let's visualize the result on an example dataset.

Given:

```
val employeeDF = sc.parallelize(...).toDF
```

```
// employeeDF:      "employees"
// +---+-----+-----+-----+
// | id|fname| lname|age|   city|
// +---+-----+-----+-----+
// | 12| Joe| Smith| 38|New York|
// |563|Sally| Owens| 48|New York|
// |645|Slate|Markham| 28| Sydney|
// |221|David| Walker| 21| Sydney|
// +---+-----+-----+-----+
```

A More Interesting SQL Query

Let's visualize the result on an example dataset.

Given:

```
val employeeDF = sc.parallelize(...).toDF
```

```
val sydneyEmployeesDF  
= spark.sql("""SELECT id, lname  
FROM employees  
WHERE city = "Sydney"  
ORDER BY id""")
```

Result ↴

```
// employeeDF:  
// +---+-----+-----+-----+  
// | id|fname| lname|age| city|  
// +---+-----+-----+---+-----+  
// | 12| Joe| Smith| 38|New York|  
// |563|Sally| Owens| 48|New York|  
// |645|Slate|Markham| 28| Sydney|  
// |221|David| Walker| 21| Sydney|  
// +---+-----+-----+-----+
```

```
sydneyEmployeesDF:  
+---+-----+  
| id| lname|  
+---+-----+  
|221| Walker|  
|645|Markham|  
+---+-----+
```

Note: it's best to use Spark 2.1+ with Scala 2.11+ for doing SQL queries with Spark SQL.



ÉCOLE POLYTECHNIQUE
FÉDÉRALE DE LAUSANNE

DataFrames (1)

Big Data Analysis with Scala and Spark

Heather Miller

DataFrames

So far, we got an intuition of what `DataFrames` are, and we learned how to create them. We also saw that if we have a `DataFrame`, we use SQL syntax and do SQL queries on them.

DataFrames have their own APIs as well!

DataFrames

So far, we got an intuition of what `DataFrames` are, and we learned how to create them. We also saw that if we have a `DataFrame`, we use SQL syntax and do SQL queries on them.

DataFrames have their own APIs as well!

In this session we'll focus on the `DataFrames` API. We'll dig into:

- ▶ available `DataFrame` data types
- ▶ some basic operations on `DataFrames`
- ▶ aggregations on `DataFrames`

DataFrames: In a Nutshell

DataFrames are...

A relational API over Spark's RDDs

Because sometimes it is more convenient to use declarative relational APIs than functional APIs for analysis jobs.

DataFrames: In a Nutshell

DataFrames are...

A relational API over Spark's RDDs

Because sometimes it is more convenient to use declarative relational APIs than functional APIs for analysis jobs.

Able to be automatically aggressively optimized

Spark SQL applies years of research on relational optimizations in the databases community to Spark.

DataFrames: In a Nutshell

DataFrames are...

A relational API over Spark's RDDs

Because sometimes it is more convenient to use declarative relational APIs than functional APIs for analysis jobs.

Able to be automatically aggressively optimized

Spark SQL applies years of research on relational optimizations in the databases community to Spark.

Untyped!

The elements within DataFrames are Rows, which are not parameterized by a type. Therefore, the Scala compiler cannot type check Spark SQL schemas in DataFrames.

DataFrames Data Types

To enable optimization opportunities, Spark SQL's DataFrames operate on a restricted set of data types.

DataFrames Data Types

To enable optimization opportunities, Spark SQL's DataFrames operate on a restricted set of data types.

Basic Spark SQL Data Types:

<i>Scala Type</i>	<i>SQL Type</i>	<i>Details</i>
Byte	ByteType	1 byte signed integers (-128,127)
Short	ShortType	2 byte signed integers (-32768,32767)
Int	IntegerType	4 byte signed integers (-2147483648,2147483647)
Long	LongType	8 byte signed integers
java.math.BigDecimal	DecimalType	Arbitrary precision signed decimals
Float	FloatType	4 byte floating point number
Double	DoubleType	8 byte floating point number
Array[Byte]	BinaryType	Byte sequence values
Boolean	BooleanType	true/false
Boolean	BooleanType	true/false
java.sql.Timestamp	TimestampType	Date containing year, month, day, hour, minute, and second.
java.sql.Date	DateType	Date containing year, month, day.
String	StringType	Character string values (stored as UTF8)

DataFrames Data Types

To enable optimization opportunities, Spark SQL's DataFrames operate on a restricted set of data types.

Complex Spark SQL Data Types:

<i>Scala Type</i>	<i>SQL Type</i>
Array[T]	ArrayType(elementType, containsNull)
Map[K, V]	MapType(keyType, valueType, valueContainsNull)
case class	StructType(List[StructFields])

DataFrames Data Types

To enable optimization opportunities, Spark SQL's DataFrames operate on a restricted set of data types.

Complex Spark SQL Data Types:

Scala Type	SQL Type
Array[T]	ArrayType(elementType, containsNull)
Map[K, V]	MapType(keyType, valueType, valueContainsNull)
case class	StructType(List[StructFields])

Arrays

Array of only one type of element (elementType). containsNull is set to true if the elements in ArrayType value can have null values.

Example:

```
// Scala type      // SQL type  
Array[String]     ArrayType(StringType, true)
```

DataFrames Data Types

To enable optimization opportunities, Spark SQL's DataFrames operate on a restricted set of data types.

Complex Spark SQL Data Types:

Scala Type SQL Type

Array[T] ArrayType(elementType, containsNull)

Map[K, V] MapType(keyType, valueType, valueContainsNull)

case class StructType(List[StructFields])

Maps

Map of key/value pairs with two types of elements. valuecontainsNull is set to true if the elements in MapType value can have null values.

Example:

// Scala type // SQL type

Map[Int, String] MapType(IntegerType, StringType, true)

DataFrames Data Types

To enable optimization opportunities, Spark SQL's DataFrames operate on a restricted set of data types.

Complex Spark SQL Data Types:

Scala Type	SQL Type
Array[T]	ArrayType(elementType, containsNull)
Map[K, V]	MapType(keyType, valueType, valueContainsNull)
case class	StructType(List[StructFields])

Structs

Struct type with list of possible fields of different types. containsNull is set to true if the elements in StructFields can have null values.

Example:

```
// Scala type // SQL type
case class Person(name: String, age: Int) StructType(List(StructField("name", StringType, true),
                                                               StructField("age", IntegerType, true)))
```

Complex Data Types Can Be Combined!

It's possible to arbitrarily nest complex data types! For example:

```
// Scala type
case class Account(
    balance: Double,
    employees:
    Array[Employee])

case class Employee(
    id: Int,
    name: String,
    jobTitle: String)

case class Project(
    title: String,
    team: Array[Employee],
    acct: Account)

// SQL type
StructType(
    StructField(title,StringType,true),
    StructField(
        team,
        ArrayType(
            StructType(StructField(id,IntegerType,true),
            StructField(name,StringType,true),
            StructField(jobTitle,StringType,true)),
            true),
            true),
    StructField(
        acct,
        StructType(
            StructField(balance[DoubleType,true],
            StructField(
                employees,
                ArrayType(
                    StructType(StructField(id,IntegerType,true),
                    StructField(name,StringType,true),
                    StructField(jobTitle,StringType,true)),
                    true),
                    true)
                ),
                true)
            )
        )
```

Accessing Spark SQL Types

Important.

In order to access *any* of these data types, either basic or complex, you must first import Spark SQL types!

```
import org.apache.spark.sql.types._
```

DataFrames Operations Are More Structured!

When introduced, the `DataFrames API` introduced a number of relational operations.

The main difference between the `RDD API` and the `DataFrames API` was that `DataFrame APIs` accept `Spark SQL expressions`, instead of arbitrary user-defined function literals like we were used to on `RDDs`. This allows the optimizer to understand what the computation represents, and for example with `filter`, it can often be used to skip reading unnecessary records.

DataFrames API: Similar-looking to SQL. Example methods include:

- ▶ `select`
- ▶ `where`
- ▶ `limit`
- ▶ `orderBy`
- ▶ `groupBy`
- ▶ `join`

Getting a look at your data

Before we get into transformations and actions on DataFrames, let's first look at the ways we can have a look at our data set.

show() pretty-prints DataFrame in tabular form. Shows first 20 elements.

Getting a look at your data

Before we get into transformations and actions on DataFrames, let's first look at the ways we can have a look at our data set.

show() pretty-prints DataFrame in tabular form. Shows first 20 elements.

Example:

```
case class Employee(id: Int, fname: String, lname: String, age: Int, city: String)
val employeeDF = sc.parallelize(...).toDF
employeeDF.show()
// +---+-----+-----+---+-----+
// | id|fname| lname|age| city|
// +---+-----+-----+---+-----+
// | 12| Joe| Smith| 38|New York|
// |563|Sally| Owens| 48|New York|
// |645|Slate|Markham| 28| Sydney|
// |221|David| Walker| 21| Sydney|
// +---+-----+-----+---+-----+
```

Getting a look at your data

Before we get into transformations and actions on DataFrames, let's first look at the ways we can have a look at our data set.

printSchema() prints the schema of your DataFrame in a tree format.

Getting a look at your data

Before we get into transformations and actions on DataFrames, let's first look at the ways we can have a look at our data set.

printSchema() prints the schema of your DataFrame in a tree format.

Example:

```
case class Employee(id: Int, fname: String, lname: String, age: Int, city: String)
val employeeDF = sc.parallelize(...).toDF
employeeDF.printschema()
```

```
// root
//   |-- id: integer (nullable = true)
//   |-- fname: string (nullable = true)
//   |-- lname: string (nullable = true)
//   |-- age: integer (nullable = true)
//   |-- city: string (nullable = true)
```

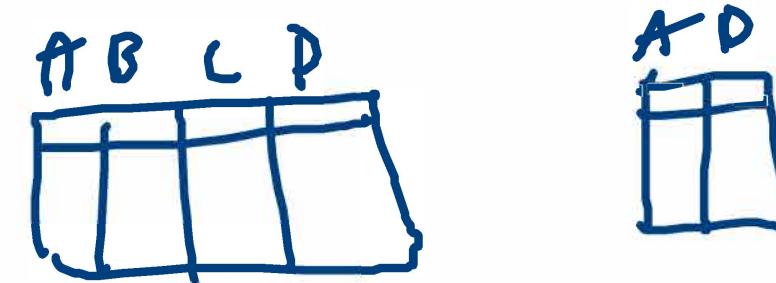
Common DataFrame Transformations

Like on RDDs, transformations on DataFrames are (1) operations which return a DataFrame as a result, and (2) are lazily evaluated.

Common DataFrame Transformations

Like on RDDs, transformations on DataFrames are (1) operations which return a DataFrame as a result, and (2) are lazily evaluated.

Some common transformations include:



```
def select(col: String, cols: String*): DataFrame
```

// selects a set of named columns and returns a new DataFrame with these
// columns as a result.

```
def agg(expr: Column, exprs: Column*): DataFrame
```

// performs aggregations on a series of columns and returns a new DataFrame
// with the calculated output.

```
def groupBy(col1: String, cols: String*): DataFrame // simplified
```

// groups the DataFrame using the specified columns. Intended to be used before an aggregation.

```
def join(right: DataFrame): DataFrame // simplified
```

// inner join with another DataFrame

Common DataFrame Transformations

Like on RDDs, transformations on DataFrames are (1) operations which return a DataFrame as a result, and (2) are lazily evaluated.

Some common transformations include:

```
def select(col: String, cols: String*): DataFrame  
// selects a set of named columns and returns a new DataFrame with these  
// columns as a result.
```

```
def agg(expr: Column, exprs: Column*): DataFrame  
// performs aggregations on a series of columns and returns a new DataFrame  
// with the calculated output.
```

```
def groupBy(col1: String, cols: String*): DataFrame // simplified  
// groups the DataFrame using the specified columns. Intended to be used before an aggregation.
```

```
def join(right: DataFrame): DataFrame // simplified  
// inner join with another DataFrame
```

Other transformations include: filter, limit, orderBy, where, as, sort, union, drop, amongst others.

Specifying Columns

As you might have observed from the previous slide, most methods take a parameter of type Column or String, always referring to some attribute/column in the data set.

Most methods on DataFrames tend to some well-understood, pre-defined operation on a column of the data set

You can select and work with columns in three ways:

1. Using \$-notation

```
// $-notation requires: import spark.implicits._  
df.filter($"age" > 18)
```

2. Referring to the Dataframe

```
df.filter(df("age") > 18))
```

sql("...")

3. Using SQL query string

```
df.filter("age > 18")
```

DataFrame Transformations: Example

Example:

Recall the example SQL query that we did in the previous session on a data set of employees. Rather than using SQL syntax, let's convert our example to use the DataFrame API.

```
case class Employee(id: Int, fname: String, lname: String, age: Int, city: String)  
val employeeDF = sc.parallelize(...).toDF
```

We'd like to obtain just the IDs and last names of employees working in a specific city, say, Sydney, Australia. Let's sort our result in order of increasing employee ID.

How could we solve this with the DataFrame API?

DataFrame Transformations: Example

Example:

We'd like to obtain just the IDs and last names of employees working in a specific city, say, Sydney, Australia. Let's sort in order of increasing employee ID.

```
case class Employee(id: Int, fname: String, lname: String, age: Int, city: String)
val employeeDF = sc.parallelize(...).toDF

val sydneyEmployeesDF = employeeDF.select("id", "lname")
    .where("city == 'Sydney'")
    .orderBy("id")
```

DataFrame Transformations: Example

Example:

We'd like to obtain just the IDs and last names of employees working in a specific city, say, Sydney, Australia. Let's sort in order of increasing employee ID.

```
case class Employee(id: Int, fname: String, lname: String, age: Int, city: String)  
val employeeDF = sc.parallelize(...).toDF
```

```
val sydneyEmployeesDF = employeeDF.select("id", "lname")  
                                .where("city == 'Sydney'")  
                                .orderBy("id")
```

```
// employeeDF:  
// +---+-----+-----+-----+  
// | id|fname| lname|age| city|  
// +---+-----+-----+-----+  
// | 12| Joe| Smith| 38|New York|  
// |563|Sally| Owens| 48|New York|  
// |645|Slate|Markham| 28| Sydney|  
// |221|David| Walker| 21| Sydney|  
// +---+-----+-----+-----+
```

DataFrame Transformations: Example

Example:

We'd like to obtain just the IDs and last names of employees working in a specific city, say, Sydney, Australia. Let's sort in order of increasing employee ID.

```
case class Employee(id: Int, fname: String, lname: String, age: Int, city: String)  
val employeeDF = sc.parallelize(...).toDF
```

```
val sydneyEmployeesDF = employeeDF.select("id", "lname")  
                                .where("city == 'Sydney'")  
                                .orderBy("id")
```

// employeeDF:					sydneyEmployeesDF:	
id	fname	lname	age	city	id	lname
12	Joe	Smith	38	New York	221	Walker
563	Sally	Owens	48	New York	645	Markham
645	Slate	Markham	28	Sydney		
221	David	Walker	21	Sydney		

Filtering in Spark SQL

The DataFrame API makes two methods available for filtering:
filter and **where** (from SQL). *They are equivalent!*

```
val over30 = employeeDF.filter("age > 30").show()  
// +---+-----+---+-----+  
// | id|fname|lname|age|    city|  
// +---+-----+---+-----+  
// | 12|  Joe|Smith| 38|New York|  
// |563|Sally|Owens| 48|New York|  
// +---+-----+---+-----+
```

```
val over30 = employeeDF.where("age > 30").show()  
// +---+-----+---+-----+  
// | id|fname|lname|age|    city|  
// +---+-----+---+-----+  
// | 12|  Joe|Smith| 38|New York|  
// |563|Sally|Owens| 48|New York|  
// +---+-----+---+-----+
```

Filtering in Spark SQL

The DataFrame API makes two methods available for filtering:
filter and **where** (from SQL). *They are equivalent!*

```
val over30 = employeeDF.filter("age > 30").show()  
// +---+-----+-----+  
// | id|fname|lname|age|    city|  
// +---+-----+-----+  
// | 12|  Joe|Smith| 38|New York|  
// |563|Sally|Owens| 48|New York|  
// +---+-----+-----+
```

```
val over30 = employeeDF.where("age > 30").show()  
// +---+-----+-----+  
// | id|fname|lname|age|    city|  
// +---+-----+-----+  
// | 12|  Joe|Smith| 38|New York|  
// |563|Sally|Owens| 48|New York|  
// +---+-----+-----+
```

Filters can be more complex too:

We can compare results between attributes/columns. Though can be more difficult to optimize.

```
employeeDF.filter($"age" > 25) &&($"city" === "Sydney").show()  
// +---+-----+-----+  
// | id|fname| lname|age| city|  
// +---+-----+-----+  
// |645|Slate|Markham| 28|Sydney|  
// +---+-----+-----+
```

Grouping and Aggregating on DataFrames

One of the most common tasks on tables is to (1) group data by a certain attribute, and then (2) do some kind of aggregation on it like a count.

Grouping and Aggregating on DataFrames

One of the most common tasks on tables is to (1) group data by a certain attribute, and then (2) do some kind of aggregation on it like a count.

For grouping & aggregating, Spark SQL provides:

- ▶ a `groupBy` function which returns a [RelationalGroupedDataset](#)
- ▶ which has several standard aggregation functions defined on it like `count`, `sum`, `max`, `min`, and `avg`.

Grouping and Aggregating on DataFrames

One of the most common tasks on tables is to (1) group data by a certain attribute, and then (2) do some kind of aggregation on it like a count.

For grouping & aggregating, Spark SQL provides:

- ▶ a groupBy function which returns a RelationalGroupedDataset
- ▶ which has several standard aggregation functions defined on it like count, sum, max, min, and avg.

How to group and aggregate?

- ▶ Just call groupBy on specific attribute/column(s) of a DataFrame,
- ▶ followed by a call to a method on RelationalGroupedDataset like count, max, or agg (for agg, also specify which attribute/column(s) subsequent spark.sql.functions like count, sum, max, etc, should be called upon.)

```
df.groupBy("attribute1")  
    .agg(sum("attribute2"))
```

```
df.groupBy("attribute1")  
    .count("attribute2")
```

Grouping and Aggregating on DataFrames: Example

Example:

Let's assume that we have a dataset of homes currently for sale in an entire US state. Let's calculate the most expensive, and least expensive homes for sale per zip code.

```
case class Listing(street: String, zip: Int, price: Int)
```

```
val listingsDF = ... // DataFrame of Listings
```

How could we do this with DataFrames?

Grouping and Aggregating on DataFrames: Example

Example:

Let's assume that we have a dataset of homes currently for sale in an entire US state. Let's calculate the most expensive, and least expensive homes for sale per zip code.

```
case class Listing(street: String, zip: Int, price: Int)
```

```
val listingsDF = ... // DataFrame of Listings
```

```
import org.apache.spark.sql.functions._
```

```
val mostExpensiveDF = listingsDF.groupBy($"zip")
    .max("price")
```

```
val leastExpensiveDF = listingsDF.groupBy($"zip")
    .min("price")
```

Grouping and Aggregating on DataFrames: Harder Example

Example:

Let's assume we have the following data set representing all of the posts in a busy open source community's Discourse forum.

```
case class Post(authorID: Int, subforum: String, likes: Int, date: String)  
  
val postsDF = ... // DataFrame of Posts
```

Let's say we would like to tally up each authors' posts per subforum, and then rank the authors with the most posts per subforum.

How could we do this with DataFrames?

Grouping and Aggregating on DataFrames: Harder Example

Example:

Let's assume we have the following data set representing all of the posts in a busy open source community's Discourse forum.

```
case class Post(authorID: Int, subforum: String, likes: Int, date: String)  
  
val postsDF = ... // DataFrame of Posts
```

Let's say we would like to tally up each authors' posts per subforum, and then rank the authors with the most posts per subforum.

```
import org.apache.spark.sql.functions._  
  
val rankedDF =  
  postsDF.groupBy($"authorID", $"subforum")  
    .agg(count($"authorID")) // new DF with columns authorID, subforum, count(authorID)  
    .orderBy($"subforum", $"count(authorID)".desc)
```

Grouping and Aggregating on DataFrames: Harder Example

Example: Let's say we would like to tally up each authors' posts per subforum, and then rank the authors with the most posts per subforums.

```
val rankedDF = postsDF.groupBy($"authorID", $"subforum")
    .agg(count($"authorID"))
    .orderBy($"subforum", $"count(authorID)".desc)

// postsDF:
// +-----+-----+-----+
// |authorID|subforum|likes|date|
// +-----+-----+-----+
// |      1| design|    2|   |
// |      1| debate|    0|   |
// |      2| debate|    0|   |
// |      3| debate|   23|   |
// |      1| design|    1|   |
// |      1| design|    0|   |
// |      2| design|    0|   |
// |      2| debate|    0|   |
// +-----+-----+-----+
```

rankedDF:

```
+-----+-----+-----+
|authorID|subforum|count(authorID)|
+-----+-----+-----+
|      2| debate|        2|
|      1| debate|        1|
|      3| debate|        1|
|      1| design|        3|
|      2| design|        1|
+-----+-----+-----+
```

The code defines a DataFrame `rankedDF` by grouping the original `postsDF` by `authorID` and `subforum`, then aggregating the count of `authorID` for each group. Finally, it orders the results by `subforum` and `count(authorID)` in descending order. The resulting `rankedDF` contains four rows, each representing a subforum and its total count of authors.

Grouping and Aggregating on DataFrames

← API

After calling groupBy, methods on RelationalGroupedDataset:

To see a list of all operations you can call following a groupBy, see the API docs for RelationalGroupedDataset.

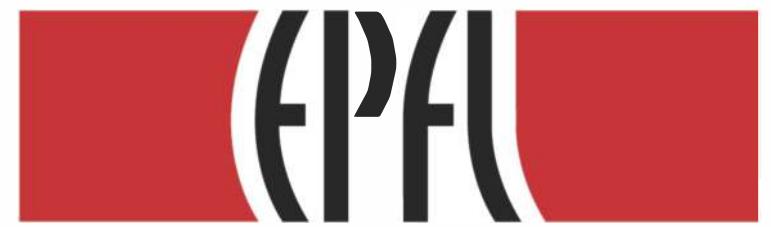
<http://spark.apache.org/docs/latest/api/scala/index.html#org.apache.spark.sql.RelationalGroupedDataset>

← API

Methods within agg:

Examples include: min, max, sum, mean, stddev, count, avg, first, last. To see a list of all operations you can call within an agg, see the API docs for org.apache.spark.sql.functions.

[http://spark.apache.org/docs/latest/api/scala/index.html#org.apache.spark.sql.functions\\$](http://spark.apache.org/docs/latest/api/scala/index.html#org.apache.spark.sql.functions$)



ÉCOLE POLYTECHNIQUE
FÉDÉRALE DE LAUSANNE

DataFrames (2)

Big Data Analysis with Scala and Spark

Heather Miller

DataFrames

So far, we got an intuition of what `DataFrames` are, how to create them, and how to do many important transformations and aggregations on them.

DataFrames

So far, we got an intuition of what `DataFrames` are, how to create them, and how to do many important transformations and aggregations on them.

In this session we'll focus on the `DataFrames` API. We'll dig into:

- ▶ working with missing values
- ▶ common actions on `DataFrames`
- ▶ joins on `DataFrames`
- ▶ optimizations on `DataFrames`

Cleaning Data with DataFrames

Sometimes you may have a data set with null or NaN values. In these cases it's often desirable to do one of the following:

- ▶ drop rows/records with unwanted values like null or "NaN"
- ▶ replace certain values with a constant

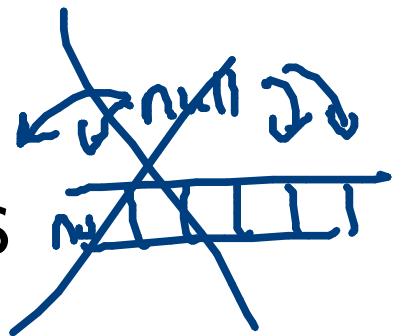
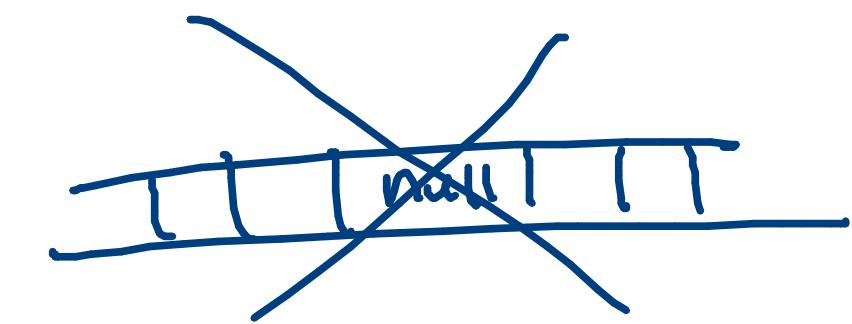
Cleaning Data with DataFrames

Sometimes you may have a data set with null or NaN values. In these cases it's often desirable to do one of the following:

- ▶ drop rows/records with unwanted values like null or "NaN"
- ▶ replace certain values with a constant

Dropping records with unwanted values:

- ▶ `drop()` drops rows that contain null or NaN values in any column and returns a new DataFrame.
- ▶ `drop("all")` drops rows that contain null or NaN values in **all** columns and returns a new DataFrame.
- ▶ `drop(Array("id", "name"))` drops rows that contain null or NaN values in the **specified** columns and returns a new DataFrame.



Cleaning Data with DataFrames

Sometimes you may have a data set with null or NaN values. In these cases it's often desirable to do one of the following:

- ▶ drop rows/records with unwanted values like null or "NaN"
- ▶ replace certain values with a constant

Replacing unwanted values:

- ▶ `fill(0)` replaces all occurrences of null or NaN in numeric columns with **specified value** and returns a new DataFrame.
- ▶ `fill(Map("minBalance" -> 0))` replaces all occurrences of null or NaN in **specified column** with **specified value** and returns a new DataFrame.
- ▶ `replace(Array("id"), Map(1234 -> 8923))` replaces **specified value** (1234) in **specified column** (id) with **specified replacement value** (8923) and returns a new DataFrame.

Common Actions on DataFrames

Like RDDs, DataFrames also have their own set of **actions**.
We've even used one several times already.

Common Actions on DataFrames

Like RDDs, DataFrames also have their own set of **actions**.
We've even used one several times already.

collect(): Array[Row]

Returns an array that contains all of Rows in this DataFrame.

count(): Long

Returns the number of rows in the DataFrame.

first(): Row/head(): Row

Returns the first row in the DataFrame.

show(): Unit

Displays the top 20 rows of DataFrame in a tabular form.

take(n: Int): Array[Row]

Returns the first n rows in the DataFrame.

Joins on DataFrames

Joins on DataFrames are similar to those on Pair RDDs, with the one major usage difference that, since DataFrames aren't key/value pairs, we have to specify which columns we should join on.

Several types of joins are available:

inner, outer, left_outer, right_outer, leftsemi.

Joins on DataFrames

Joins on DataFrames are similar to those on Pair RDDs, with the one major usage difference that, since DataFrames aren't key/value pairs, we have to specify which columns we should join on.

Several types of joins are available:

inner, outer, left_outer, right_outer, leftsemi.

Performing joins:

Given two DataFrames, df1 and df2 each with a column/attribute called id, we can perform an inner join as follows:

```
df1.join(df2,   $$df1.id" === $$df2.id")
```

It's possible to change the join type by passing an additional string parameter to join specifying which type of join to perform. E.g.,

```
df1.join(df2,   $$df1.id" === $$df2.id", "right_outer")
```

Joins on DataFrames: A Familiar Example

Example:

Recall our CFF data set from earlier in the course. Let's adapt it to the DataFrame API.

Joins on DataFrames: A Familiar Example

Example:

Recall our CFF data set from earlier in the course. Let's adapt it to the DataFrame API.

```
case class Abo(id: Int, v: (String, String))
case class Loc(id: Int, v: String)

val as = List(Abo(101, ("Ruetli", "AG")), Abo(102, ("Brelaz", "DemiTarif")),
              Abo(103, ("Gress", "DemiTarifVisa")), Abo(104, ("Schatten", "DemiTarif")))
val abosDF = sc.parallelize(as).toDF

val ls = List(Loc(101, "Bern"), Loc(101, "Thun"), Loc(102, "Lausanne"), Loc(102, "Geneve"),
              Loc(102, "Nyon"), Loc(103, "Zurich"), Loc(103, "St-Gallen"), Loc(103, "Chur"))
val locationsDF = sc.parallelize(ls).toDF
```

Joins on DataFrames: A Familiar Example

Example:

Recall our CFF data set from earlier in the course. Let's adapt it to the DataFrame API.

```
// abosDF:  
// +---+-----+  
// | id|          v|  
// +---+-----+  
// |101|      [Ruetli,AG]|  
// |102|  [Brelaz,DemiTarif]|  
// |103|[Gress,DemiTarifV...|  
// |104|[Schatten,DemiTarif]|  
// +---+-----+  
  
//  
//  
//  
//  
// locationsDF:  
// +---+-----+  
// | id|          v|  
// +---+-----+  
// |101|      Bern|  
// |101|      Thun|  
// |102| Lausanne|  
// |102| Geneve|  
// |102| Nyon|  
// |103| Zurich|  
// |103| St-Gallen|  
// |103| Chur|  
// +---+-----+
```

Joins on DataFrames: A Familiar Example

Example:

Recall our CFF data set from earlier in the course.

How do we combine only customers that have a subscription and where there is location info?

Joins on DataFrames: A Familiar Example

Example:

Recall our CFF data set from earlier in the course.

How do we combine only customers that have a subscription and where there is location info?

We perform an inner join, of course.

```
val abosDF = sc.parallelize(as).toDF  
val locationsDF = sc.parallelize(ls).toDF
```

```
val trackedCustomersDF =  
    abosDF.join(locationsDF, abosDF("id") === locationsDF("id"))
```

Joins on DataFrames: A Familiar Example

Example:

How do we combine only customers that have a subscription and where there is location info?

```
val trackedCustomersDF =  
    abosDF.join(locationsDF, abosDF("id") === locationsDF("id"))
```

```
// trackedCustomersDF:  
// +---+-----+---+-----+  
// | id|          v| id|      v|  
// +---+-----+---+-----+  
// |101|      [Ruetli,AG]|101|    Bern|  
// |101|      [Ruetli,AG]|101|    Thun|  
// |103|[Gress,DemiTarifV...|103| Zurich|  
// |103|[Gress,DemiTarifV...|103|St-Gallen|  
// |103|[Gress,DemiTarifV...|103|    Chur|  
// |102| [Brelaz,DemiTarif]|102| Lausanne|  
// |102| [Brelaz,DemiTarif]|102|   Geneve|  
// |102| [Brelaz,DemiTarif]|102|     Nyon|  
// +---+-----+---+-----+
```

Joins on DataFrames: A Familiar Example

Example:

How do we combine only customers that have a subscription and where there is location info?

```
val trackedCustomersDF =  
    abosDF.join(locationsDF, abosDF("id") === locationsDF("id"))  
  
// trackedCustomersDF:  
// +---+-----+---+-----+  
// | id|          v| id|      v|  
// +---+-----+---+-----+  
// |101| [Ruetli,AG]|101|   Bern|  
// |101| [Ruetli,AG]|101|   Thun|  
// |103|[Gress,DemiTarifV...|103| Zurich|  
// |103|[Gress,DemiTarifV...|103|St-Gallen|  
// |103|[Gress,DemiTarifV...|103|   Chur|  
// |102| [Brelaz,DemiTarif]|102| Lausanne|  
// |102| [Brelaz,DemiTarif]|102| Geneve|  
// |102| [Brelaz,DemiTarif]|102|   Nyon|  
// +---+-----+---+-----+
```

As expected, customer 104 is missing! :-)

Joins on DataFrames: A Familiar Example

Example: Let's assume the CFF wants to know for which subscribers the CFF has managed to collect location information. E.g., it's possible that someone has a demi-tarif, but doesn't use the CFF app and only pays cash for tickets.

Which join do we use?

Joins on DataFrames: A Familiar Example

Example: Let's assume the CFF wants to know for which subscribers the CFF has managed to collect location information. E.g., it's possible that someone has a demi-tarif, but doesn't use the CFF app and only pays cash for tickets.

```
val abosWithOptionalLocationsDF
  = abosDF.join(locationsDF, abosDF("id") === locationsDF("id"), "left_outer")
// +-----+-----+
// | id|      v|  id|      v|
// +-----+-----+
// |101| [Ruetli,AG]| 101|    Bern|
// |101| [Ruetli,AG]| 101|    Thun|
// |103|[Gress,DemiTarifV...| 103| Zurich|
// |103|[Gress,DemiTarifV...| 103|St-Gallen|
// |103|[Gress,DemiTarifV...| 103|    Chur|
// |102| [Brelaz,DemiTarif]| 102| Lausanne|
// |102| [Brelaz,DemiTarif]| 102|   Geneve|
// |102| [Brelaz,DemiTarif]| 102|    Nyon|
// |104|[Schatten,DemiTarif]|null|    null|
// +-----+-----+
```

Joins on DataFrames: A Familiar Example

Example: Let's assume the CFF wants to know for which subscribers the CFF has managed to collect location information. E.g., it's possible that someone has a demi-tarif, but doesn't use the CFF app and only pays cash for tickets.

```
val abosWithOptionalLocationsDF
  = abosDF.join(locationsDF, abosDF("id") === locationsDF("id"), "left_outer")
// +-----+-----+
// | id|      v|  id|      v|
// +-----+-----+
// |101| [Ruetli,AG]| 101|    Bern|
// |101| [Ruetli,AG]| 101|    Thun|
// |103|[Gress,DemiTarifV...| 103| Zurich|
// |103|[Gress,DemiTarifV...| 103|St-Gallen|
// |103|[Gress,DemiTarifV...| 103|    Chur|
// |102| [Brelaz,DemiTarif]| 102| Lausanne|
// |102| [Brelaz,DemiTarif]| 102|   Geneve|
// |102| [Brelaz,DemiTarif]| 102|    Nyon|
// |104|[Schatten,DemiTarif]|null|    null|
// +-----+-----+
```

As expected, customer 104 has returned! :-)

Revisiting Our Selecting Scholarship Recipients Example

Now that we're familiar with the DataFrames API, let's revisit the example that we looked at at a few sessions back.

Revisiting Our Selecting Scholarship Recipients Example

Now that we're familiar with the DataFrames API, let's revisit the example that we looked at at a few sessions back.

Recall Let's imagine that we are an organization, CodeAward, offering scholarships to programmers who have overcome adversity. Let's say we have the following two datasets.

```
case class Demographic(id: Int,  
                      age: Int,  
                      codingBootcamp: Boolean,  
                      country: String,  
                      gender: String,  
                      isEthnicMinority: Boolean,  
                      servedInMilitary: Boolean)  
  
val demographicsDF = sc.textfile(...).toDF // DataFrame of Demographic  
  
case class Finances(id: Int,  
                     hasDebt: Boolean,  
                     hasFinancialDependents: Boolean,  
                     hasStudentLoans: Boolean,  
                     income: Int)  
  
val financesDF = sc.textfile(...).toDF // DataFrame of Finances
```

Revisiting Our Selecting Scholarship Recipients Example

Our data sets include students from many countries, with many life and financial backgrounds. Now, let's imagine that our goal is to tally up and select students for a specific scholarship.

Revisiting Our Selecting Scholarship Recipients Example

Our data sets include students from many countries, with many life and financial backgrounds. Now, let's imagine that our goal is to tally up and select students for a specific scholarship.

As an example, Let's count:

- ▶ Swiss students
- ▶ who have debt & financial dependents

How might we implement this program with the DataFrame API?

```
// Remember, DataFrames available to us:  
val demographicsDF = sc.textfile(...).toDF // DataFrame of Demographic  
val financesDF = sc.textfile(...).toDF      // DataFrame of Finances
```

Revisiting Our Selecting Scholarship Recipients Example

With DataFrames:

```
demographicsDF.join(financesDF, demographicsDF("ID") === financesDF("ID"), "inner")
    .filter($"HasDebt" && $"HasFinancialDependents")
    .filter($"CountryLive" === "Switzerland")
    .count
```

Revisiting Our Selecting Scholarship Recipients Example

Recall

While for all three of these possible examples, the end result is the same, the time it takes to execute the job is vastly different.

Possibility 1

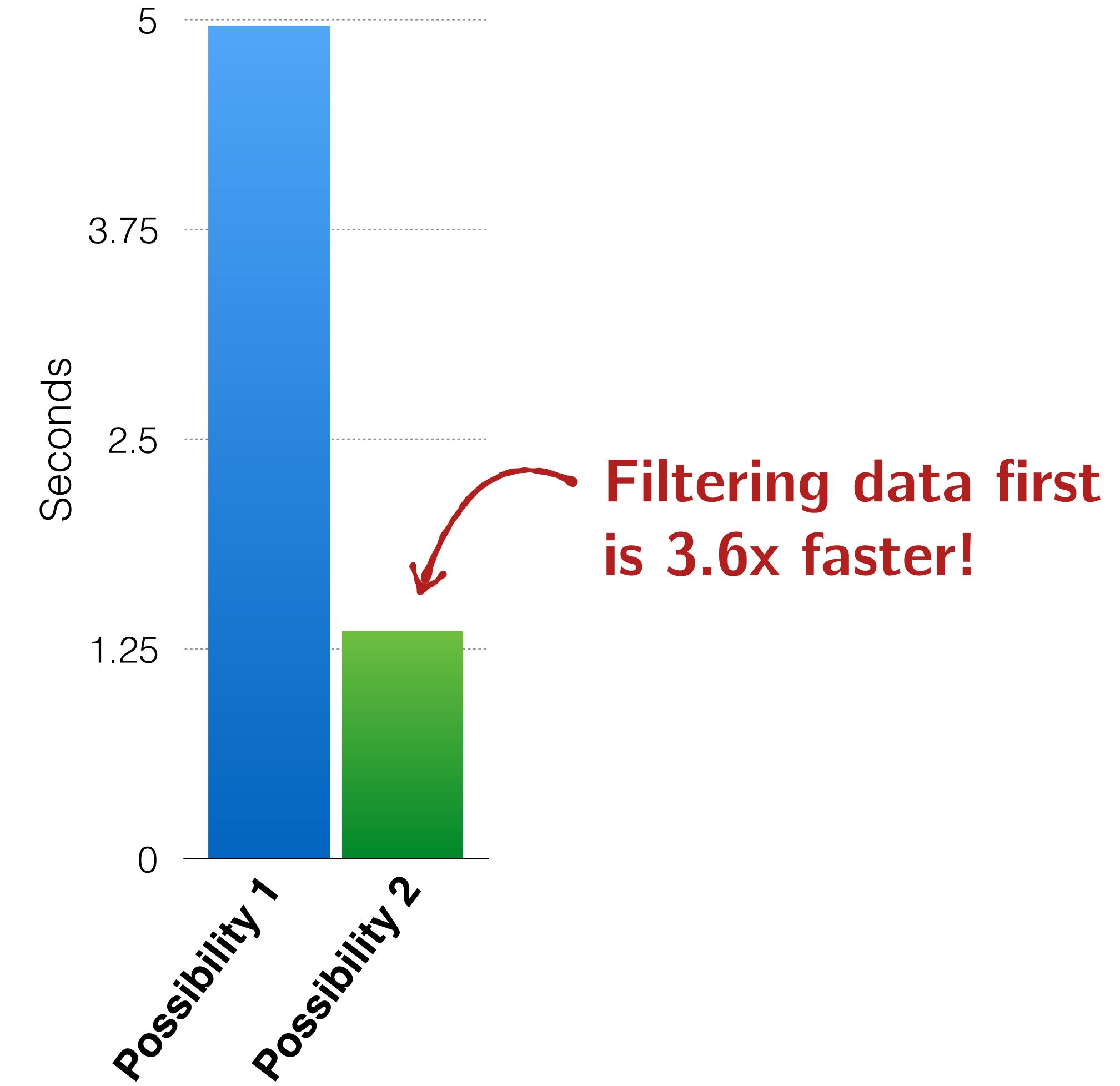
```
> ds.join(fs)
   .filter(p => p._2._
   .count

▶ (1) Spark Jobs
res0: Long = 10
Command took 4.97 seconds -
```

Possibility 2

```
> val fsi = fs.filter(
   ds.filter(p => p._2.
   .join(fsi)
   .count

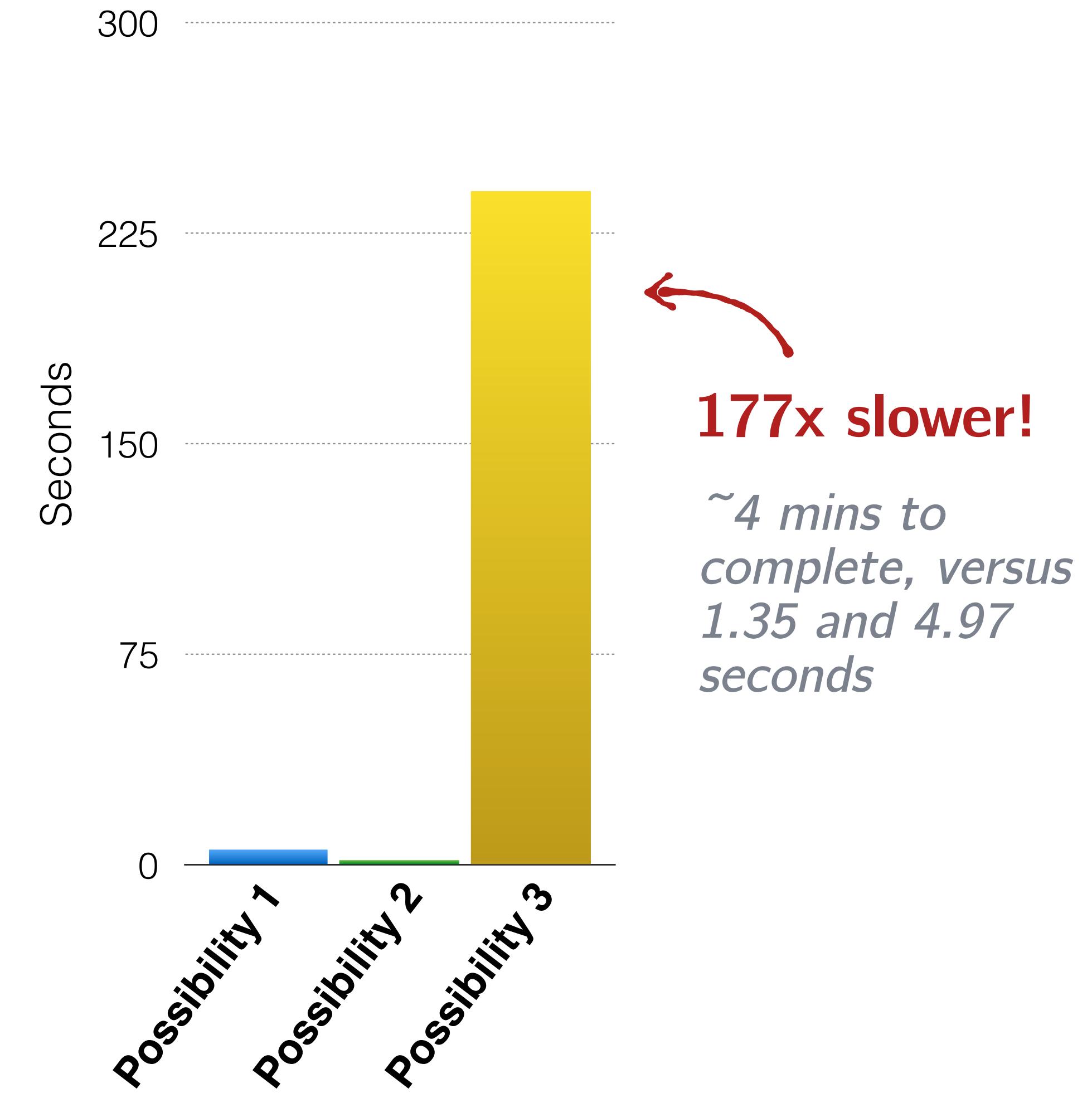
▶ (1) Spark Jobs
fsi: org.apache.spark.
res4: Long = 10
Command took 1.35 seconds -
```



Revisiting Our Selecting Scholarship Recipients Example

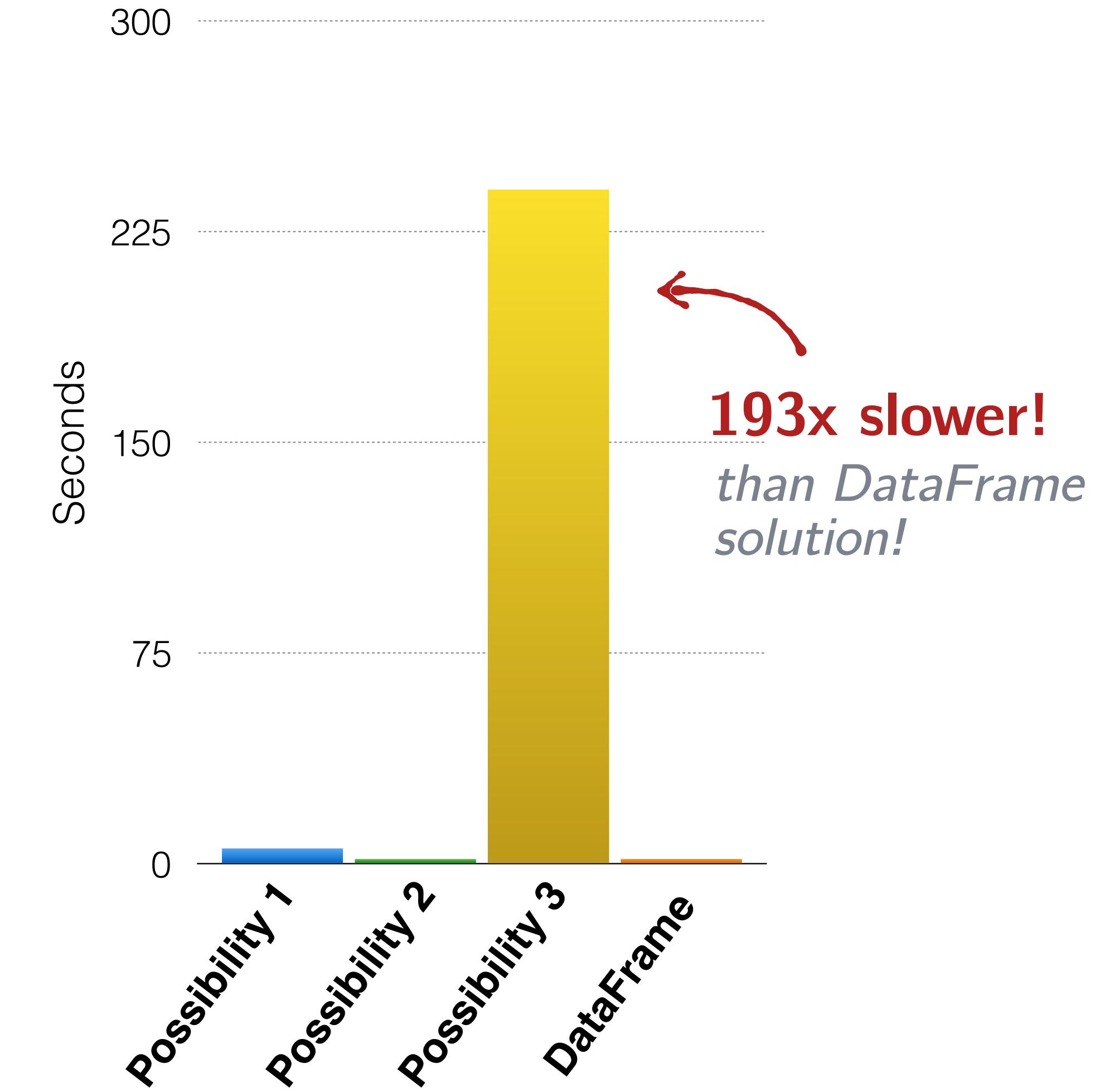
Recall

While for all three of these possible examples, the end result is the same, the time it takes to execute the job is vastly different.



Revisiting Our Selecting Scholarship Recipients Example

Comparing performance between handwritten RDD-based solutions and DataFrame solution...



Revisiting Our Selecting Scholarship Recipients Example

Comparing performance between handwritten RDD-based solutions and DataFrame solution...

Possibility 1

```
> ds.join(fs)
   .filter(p => p._2 >
   .count
```

▶ (1) Spark Jobs
res0: Long = 10
Command took 4.97 seconds -

Possibility 2

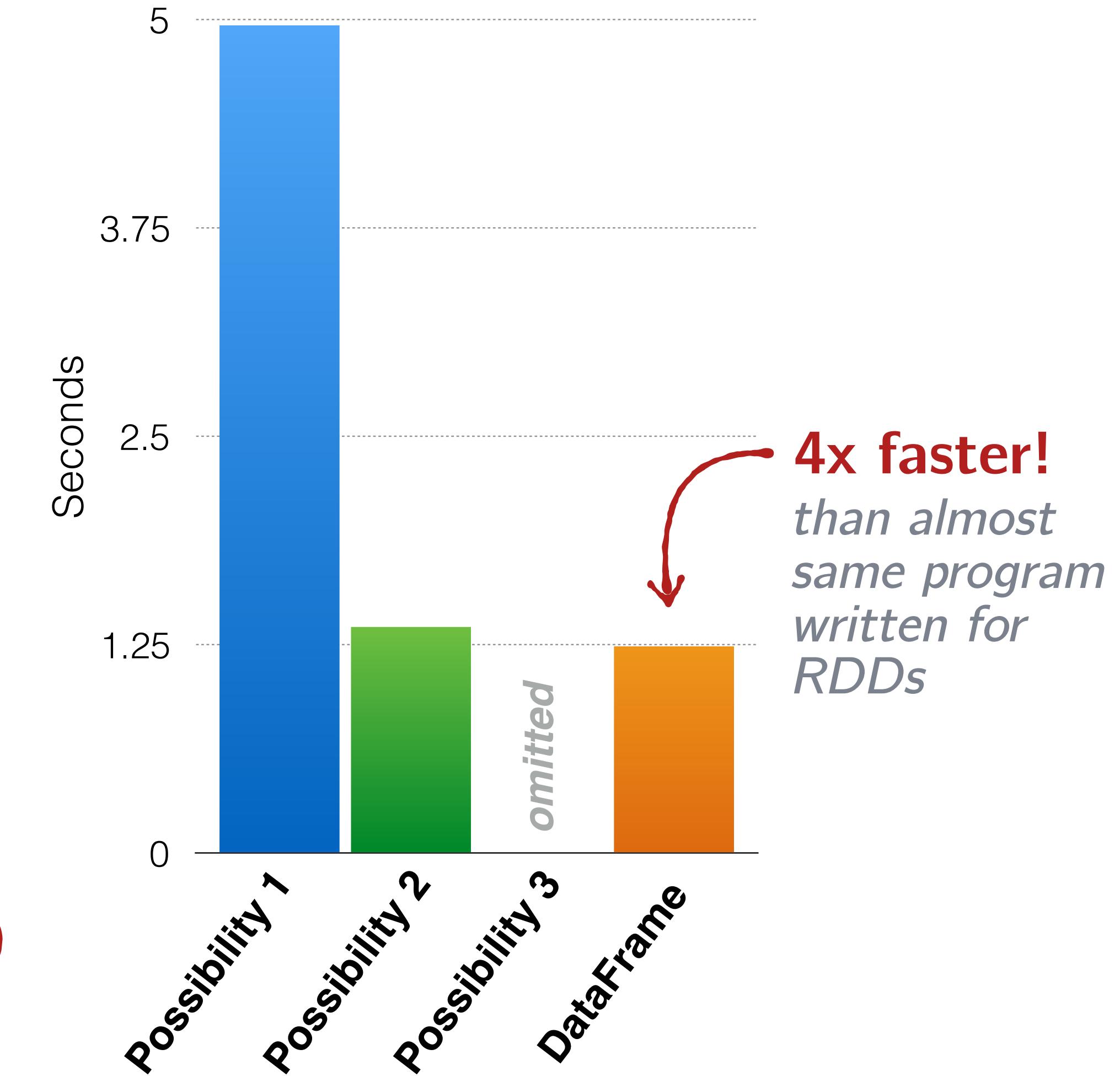
```
> val fsi = fs.filter(
  ds.filter(p => p._2 >
    .join(fsi)
    .count
```

▶ (1) Spark Jobs
fsi: org.apache.spark.
res4: Long = 10
Command took 1.35 seconds

DataFrame

```
> demographics.join(fi
   .filter(
     .filter(
       .count
```

▶ (2) Spark Jobs
res24: Long = 10
Command took 1.24 seconds -



Optimizations

How is this possible?

Optimizations

How is this possible?

Recall that Spark SQL comes with two specialized backend components:

- ▶ **Catalyst**, query optimizer.
- ▶ **Tungsten**, off-heap serializer.

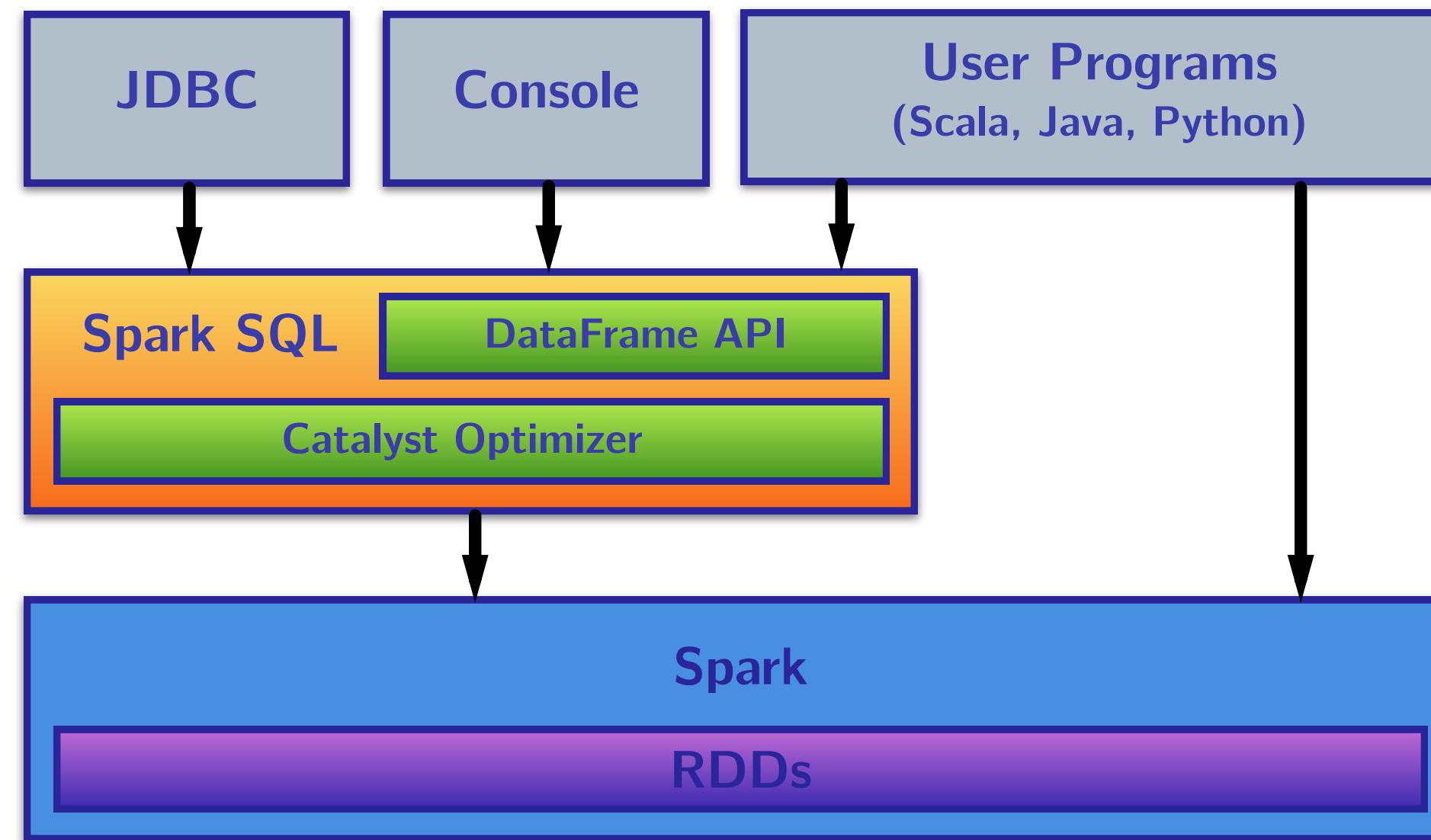
Let's briefly develop some intuition about why structured data and computations enable these two backend components to do so many optimizations for you.

Optimizations

Catalyst

Spark SQL's query optimizer.

Recall our earlier map of how Spark SQL relates to the rest of Spark:

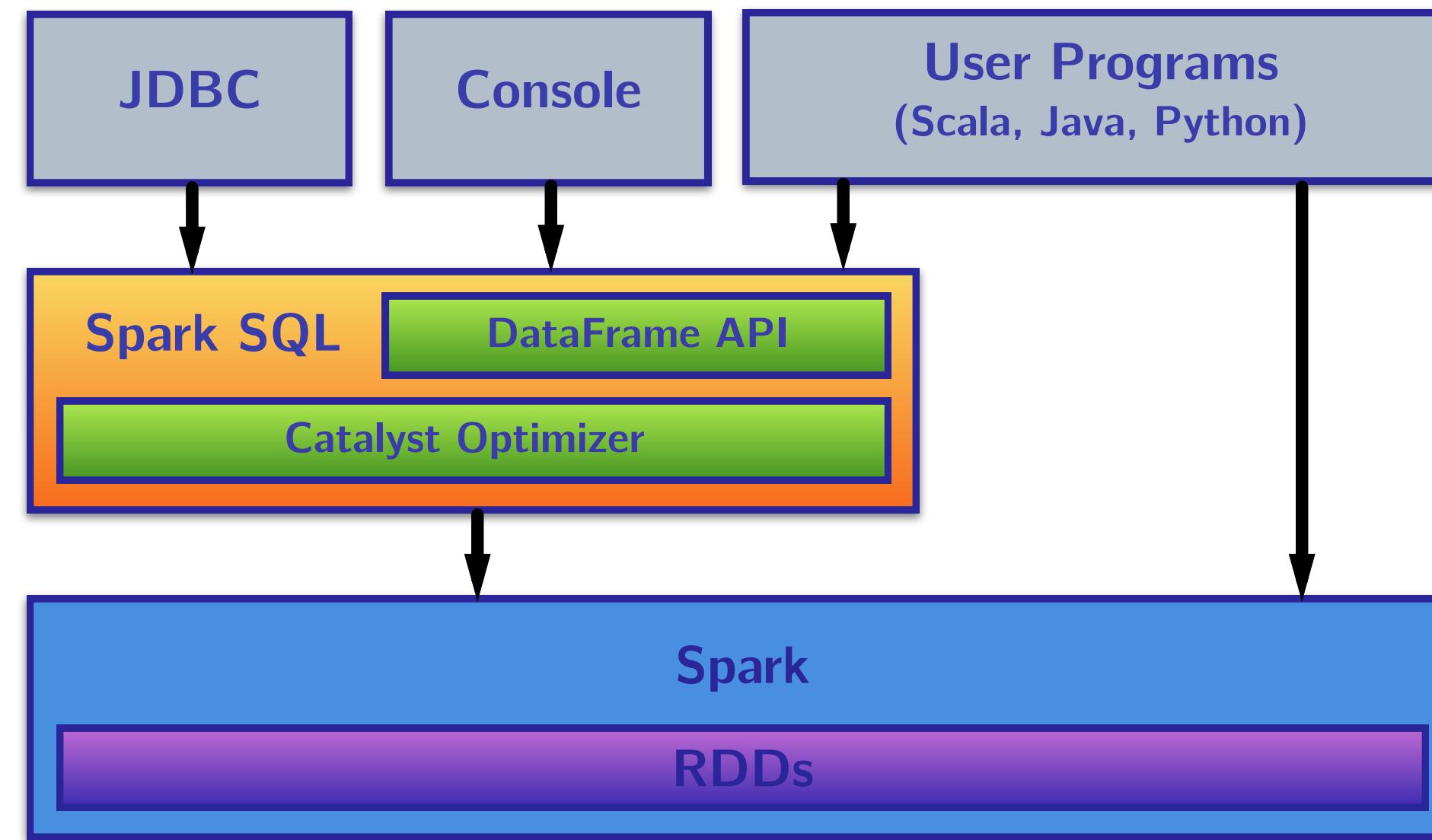


Optimizations

Catalyst

Spark SQL's query optimizer.

Recall our earlier map of how Spark SQL relates to the rest of Spark:



Key thing to remember:

Catalyst compiles Spark SQL programs down to an RDD.

Optimizations: RDDs vs DataFrames

In summary:

Spark RDDs:



Not much structure.
Difficult to aggressively optimize.

DataFrames/Databases/Hive:

name: String	balance: Double	risk: Boolean
name: String	balance: Double	risk: Boolean
name: String	balance: Double	risk: Boolean
name: String	balance: Double	risk: Boolean

SELECT
WHERE
ORDER BY
GROUP BY
COUNT

Lots of structure.
Lots of optimization opportunities!

Optimizations

Catalyst

Spark SQL's query optimizer.

Assuming Catalyst...

- ▶ has full knowledge and understanding of all data types
- ▶ knows the exact schema of our data
- ▶ has detailed knowledge of the computations we'd like to do

Optimizations

Catalyst

Spark SQL's query optimizer.

Assuming Catalyst...

- ▶ has full knowledge and understanding of all data types
- ▶ knows the exact schema of our data
- ▶ has detailed knowledge of the computations we'd like to do

Makes it possible for us to do optimizations like:

- ▶ **Reordering operations.**

Laziness + structure gives us the ability to analyze and rearrange DAG of computation/the logical operations the user would like to do, before they're executed.

E.g., Catalyst can decide to rearrange and fuse together filter operations, pushing all filters early as possible, so expensive operations later are done on less data.

Optimizations

Catalyst

Spark SQL's query optimizer.

Assuming Catalyst...

- ▶ has full knowledge and understanding of all data types
- ▶ knows the exact schema of our data
- ▶ has detailed knowledge of the computations we'd like to do

Makes it possible for us to do optimizations like:

- ▶ **Reordering operations.**
- ▶ **Reduce the amount of data we must read.**

Skip reading in, serializing, and sending around parts of the data set that aren't needed for our computation.

E.g., Imagine a Scala object containing many fields unnecessary to our computation. Catalyst can narrow down and select, serialize, and send around only relevant columns of our data set.

Optimizations

Catalyst

Spark SQL's query optimizer.

Assuming Catalyst...

- ▶ has full knowledge and understanding of all data types
- ▶ knows the exact schema of our data
- ▶ has detailed knowledge of the computations we'd like to do

Makes it possible for us to do optimizations like:

- ▶ **Reordering operations.**
- ▶ **Reduce the amount of data we must read.**
- ▶ **Pruning unneeded partitioning.**

Analyze DataFrame and filter operations to figure out and skip partitions that are unneeded in our computation.

Optimizations

Tungsten

Spark SQL's off-heap data encoder.

Since our data types are restricted to Spark SQL data types, Tungsten can provide:

- ▶ highly-specialized data encoders
- ▶ **column-based**
- ▶ off-heap (free from garbage collection overhead!)

Optimizations

Tungsten

Spark SQL's off-heap data encoder.

Since our data types are restricted to Spark SQL data types, Tungsten can provide:

- ▶ highly-specialized data encoders
- ▶ **column-based**
- ▶ off-heap (free from garbage collection overhead!)

Highly-specialized data encoders.

Tungsten can take schema information and tightly pack serialized data into memory. This means more data can fit in memory, and faster serialization/deserialization (CPU bound task)

Optimizations

Tungsten

Spark SQL's off-heap data encoder.

Since our data types are restricted to Spark SQL data types, Tungsten can provide:

- ▶ highly-specialized data encoders
- ▶ **column-based**
- ▶ off-heap (free from garbage collection overhead!)

Column-based

Based on the observation that most operations done on tables tend to be focused on specific columns/attributes of the data set. Thus, when storing data, group data by column instead of row for faster lookups of data associated with specific attributes/columns.

Well-known to be more efficient across DBMS.

Optimizations

Tungsten

Spark SQL's off-heap data encoder.

Since our data types are restricted to Spark SQL data types, Tungsten can provide:

- ▶ highly-specialized data encoders
- ▶ **column-based**
- ▶ off-heap (free from garbage collection overhead!)

Off-heap

Regions of memory off the heap, manually managed by Tungsten, so as to avoid garbage collection overhead and pauses.

Optimizations

Taken together, Catalyst and Tungsten offer ways to significantly speed up your code, even if you write it inefficiently initially.

Limitations of DataFrames

Limitations of DataFrames

Untyped!

```
listingsDF.filter($"state" === "CA")
```

```
// org.apache.spark.sql.AnalysisException:  
//cannot resolve 'state' given input columns: [street, zip, price];;
```

Your code compiles, but you get runtime exceptions when you attempt to run a query on a column that doesn't exist.

Would be nice if this was caught at compile time like we're used to in Scala!

Limitations of DataFrames

Limited Data Types

If your data can't be expressed by case classes/Products and standard Spark SQL data types, it may be difficult to ensure that a Tungsten encoder exists for your data type.

E.g., you have an application which already uses some kind of complicated regular Scala class.

Limitations of DataFrames

Limited Data Types

If your data can't be expressed by case classes/Products and standard Spark SQL data types, it may be difficult to ensure that a Tungsten encoder exists for your data type.

E.g., you have an application which already uses some kind of complicated regular Scala class.

Requires Semi-Structured/Structured Data

If your unstructured data cannot be reformulated to adhere to some kind of schema, it would be better to use RDDs.



ÉCOLE POLYTECHNIQUE
FÉDÉRALE DE LAUSANNE

Datasets

Big Data Analysis with Scala and Spark

Heather Miller

Example

Let's say we've just done the following computation on a DataFrame representing a data set of Listings of homes for sale; we've computed the average price of for sale per zipcode.

```
case class Listing(street: String, zip: Int, price: Int)
val listingsDF = ... // DataFrame of Listings

import org.apache.spark.sql.functions._
val averagePricesDF = listingsDF.groupBy($"zip")
    .avg("price")
```

Example

Let's say we've just done the following computation on a DataFrame representing a data set of Listings of homes for sale; we've computed the average price of for sale per zipcode.

```
case class Listing(street: String, zip: Int, price: Int)
val listingsDF = ... // DataFrame of Listings

import org.apache.spark.sql.functions._

val averagePricesDF = listingsDF.groupBy($"zip")
    .avg("price")
```

Great. Now let's call `collect()` on `averagePricesDF` to bring it back to the master node...

Example

```
val averagePrices = averagePricesDF.collect()  
// averagePrices: Array[org.apache.spark.sql.Row]
```

Oh no. What is this? What's in this Row thing again?

Example

```
val averagePrices = averagePricesDF.collect()  
// averagePrices: Array[org.apache.spark.sql.Row]
```

Oh no. What is this? What's in this Row thing again?

Oh right, I have to cast things because Rows don't have type information associated with them. How many columns were my result again? And what were their types?

```
val averagePricesAgain = averagePrices.map {  
    row => (row(0).asInstanceOf[String], row(1).asInstanceOf[Int])  
}
```

Example

```
val averagePrices = averagePricesDF.collect()  
// averagePrices: Array[org.apache.spark.sql.Row]
```

Oh no. What is this? What's in this Row thing again?

Oh right, I have to cast things because Rows don't have type information associated with them. How many columns were my result again? And what were their types?

```
val averagePricesAgain = averagePrices.map {  
    row => (row(0).asInstanceOf[String], row(1).asInstanceOf[Int])  
}
```

Nope.

```
// java.lang.ClassCastException
```

Example

Let's try to see what's in this Row thing. (Consults Row API docs.)

```
averagePrices.head.schema.printTreeString()  
// root  
// |-- zip: integer (nullable = true)  
// |-- avg(price): double (nullable = true)
```

Example

Let's try to see what's in this Row thing. (Consults Row API docs.)

```
averagePrices.head.schema.printTreeString()  
// root  
// |-- zip: integer (nullable = true)  
// |-- avg(price): double (nullable = true)
```

Trying again...

```
val averagePricesAgain = averagePrices.map {  
    row => (row(0).asInstanceOf[Int], row(1).asInstanceOf[Double]) // Ew...  
}  
// mostExpensiveAgain: Array[(Int, Double)]
```

yay! 🎉

Example

Let's try to see what's in this Row thing. (Consults Row API docs.)

```
averagePrices.head.schema.printTreeString()  
// root  
// |-- zip: integer (nullable = true)  
// |-- avg(price): double (nullable = true)
```

Trying again...

.price

```
val averagePricesAgain = averagePrices.map {  
    row => (row(0).asInstanceOf[Int], row(1).asInstanceOf[Double]) // Ew...  
}  
// mostExpensiveAgain: Array[(Int, Double)]
```

yay! 🎉

Wouldn't it be nice if we could have both Spark SQL optimizations and typesafety?

Datasets

Enter Datasets.



Confession

I've been keeping something from you...

DataFrames are Datasets!

DataFrames are actually Datasets.

```
type DataFrame = Dataset[Row]
```

DataFrames are Datasets!

DataFrames are actually Datasets.

```
type DataFrame = Dataset[Row]
```



What the heck is a Dataset?

DataFrames are Datasets!

DataFrames are actually Datasets.

```
type DataFrame = Dataset[Row]
```

🧐 What the heck is a Dataset?

- ▶ Datasets can be thought of as **typed** distributed collections of data.
- ▶ Dataset API unifies the DataFrame and RDD APIs. Mix and match! ↗ ↘ ↙
- ▶ Datasets require structured/semi-structured data. Schemas and Encoders core part of Datasets.

Think of Datasets as a compromise between RDDs & DataFrames.

You get more type information on Datasets than on DataFrames, and you get more optimizations on Datasets than you get on RDDs.

DataFrames are Datasets!

Example:

Let's calculate the average home price per zipcode with Datasets.

Assuming `listingsDS` is of type `Dataset[Listing]`:

```
    ↴λ
listingsDS.groupByKey(l => l.zip)      // looks like groupByKey on RDDs!
    .agg(avg($"price").as[Double]) // looks like our DataFrame operators!
```

We can freely mix APIs!

Datasets

Datasets are a something in the middle between DataFrames and RDDs

- ▶ You can still use relational DataFrame operations as we learned in previous sessions on Datasets.
- ▶ Datasets add more *typed* operations that can be used as well.
- ▶ Datasets let you use higher-order functions like map, flatMap, filter again!

Datasets

Datasets are something in the middle between DataFrames and RDDs

- ▶ You can still use relational DataFrame operations as we learned in previous sessions on Datasets.
- ▶ Datasets add more *typed* operations that can be used as well.
- ▶ Datasets let you use higher-order functions like map, flatMap, filter again!

Datasets can be used when you want a mix of functional and relational transformations while benefiting from some of the optimizations on DataFrames.

And we've *almost* got a type safe API as well.

Creating Datasets

From a DataFrame.

Just use the `toDS` convenience method.

```
myDF.toDS // requires import spark.implicits._
```

Note that often it's desirable to read in data from JSON from a file, which can be done with the `read` method on the `SparkSession` object like we saw in previous sessions, and then converted to a Dataset:

```
val myDS = spark.read.json("people.json").as[Person]
```

Creating Datasets

From a DataFrame.

Just use the `toDS` convenience method.

```
myDF.toDS // requires import spark.implicits._
```

Note that often it's desirable to read in data from JSON from a file, which can be done with the `read` method on the `SparkSession` object like we saw in previous sessions, and then converted to a Dataset:

```
val myDS = spark.read.json("people.json").as[Person]
```

From an RDD.

Just use the `toDS` convenience method.

```
myRDD.toDS // requires import spark.implicits._
```

Creating Datasets

From a DataFrame.

Just use the `toDS` convenience method.

```
myDF.toDS // requires import spark.implicits._
```

Note that often it's desirable to read in data from JSON from a file, which can be done with the `read` method on the `SparkSession` object like we saw in previous sessions, and then converted to a Dataset:

```
val myDS = spark.read.json("people.json").as[Person]
```

From an RDD.

Just use the `toDS` convenience method.

```
myRDD.toDS // requires import spark.implicits._
```

From common Scala types.

Just use the `toDS` convenience method.

```
List("yay", "ohnoes", "hooray!").toDS // requires import spark.implicits._
```

Typed Columns

Recall the Column type from DataFrames. On Datasets, *typed* operations tend to act on TypedColumn instead.

```
<console>:58: error: type mismatch;  
  found   : org.apache.spark.sql.Column  
 required: org.apache.spark.sql.TypedColumn[...]  
          .agg(avg($"price")).show  
                      ^
```

Typed Columns

Recall the Column type from DataFrames. On Datasets, *typed* operations tend to act on TypedColumn instead.

```
<console>:58: error: type mismatch;  
  found   : org.apache.spark.sql.Column  
 required: org.apache.spark.sql.TypedColumn[...]  
          .agg(avg($"price")).show  
                      ^
```

To create a TypedColumn, all you have to do is call as[...] on your (untyped) Column.

```
$"price".as[Double] // this now represents a TypedColumn.
```

Transformations on Datasets

Remember *untyped transformations* from DataFrames?

Transformations on Datasets

Remember *untyped transformations* from DataFrames?

The Dataset API includes both untyped and typed transformations.

- ▶ **untyped transformations** the transformations we learned on DataFrames.
- ▶ **typed transformations** typed variants of many DataFrame transformations + additional transformations such as RDD-like higher-order functions `map`, `flatMap`, etc.

Transformations on Datasets

Remember *untyped transformations* from DataFrames?

The Dataset API includes both untyped and typed transformations.

- ▶ **untyped transformations** the transformations we learned on DataFrames.
- ▶ **typed transformations** typed variants of many DataFrame transformations + additional transformations such as RDD-like higher-order functions `map`, `flatMap`, etc.

These APIs are integrated. You can call a `map` on a DataFrame and get back a Dataset, for example.

Caveat: not every operation you know from RDDs are available on Datasets, and not all operations look 100% the same on Datasets as they did on RDDs.

But remember, you may have to explicitly provide type information when going from a DataFrame to a Dataset via typed transformations.

```
val keyValuesDF = List((3, "Me"), (1, "Thi"), (2, "Se"), (3, "ssa"), (3, "-")), (2, "cre"), (2, "t")).toDF  
val res = keyValuesDF.map(row => row(0).asInstanceOf[Int] + 1) // Ew...
```

Common (Typed) Transformations on Datasets

map

map[U](f: T => U): Dataset[U]

Apply function to each element in the Dataset and return a Dataset of the result.

flatMap

flatMap[U](f: T => TraversableOnce[U]): Dataset[U]

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

filter

filter(pred: T => Boolean): Dataset[T]

Apply predicate function to each element in the Dataset and return a Dataset of elements that have passed the predicate condition, pred.

distinct

distinct(): Dataset[T]

Return Dataset with duplicates removed.

Common (Typed) Transformations on Datasets

groupByKey	groupByKey[K](f: T => K): KeyValueGroupedDataset[K, T] Apply function to each element in the Dataset and return a Dataset of the result.
coalesce	coalesce(numPartitions: Int): Dataset[T] Apply a function to each element in the Dataset and return a Dataset of the contents of the iterators returned.
repartition	repartition(numPartitions: Int): Dataset[T] Apply predicate function to each element in the Dataset and return a Dataset of elements that have passed the predicate condition, pred.

Grouped Operations on Datasets

Like on DataFrames, Datasets have a special set of aggregation operations meant to be used after a call to `groupByKey` on a Dataset.

- ▶ calling `groupByKey` on a Dataset returns a KeyValueGroupedDataset
- ▶ `KeyValueGroupedDataset` contains a number of aggregation operations which return Datasets

Grouped Operations on Datasets

Like on DataFrames, Datasets have a special set of aggregation operations meant to be used after a call to `groupByKey` on a Dataset.

- ▶ calling `groupByKey` on a Dataset returns a `KeyValueGroupedDataset`
- ▶ `KeyValueGroupedDataset` contains a number of aggregation operations which return Datasets

How to group & aggregate on Datasets?

1. Call `groupByKey` on a Dataset, get back a `KeyValueGroupedDataset`.
2. Use an aggregation operation on `KeyValueGroupedDataset` (return Datasets)

Note: using `groupBy` on a Dataset, you will get back a `RelationalGroupedDataset` whose aggregation operators will return a DataFrame. Therefore, be careful to avoid `groupBy` if you would like to stay in the Dataset API.

Some KeyValueGroupedDataset Aggregation Operations

reduceGroups	reduceGroups(f: (V, V) => V): Dataset[(K, V)]
	Reduces the elements of each group of data using the specified binary function. The given function must be commutative and associative or the result may be non-deterministic.
agg	agg[U](col: TypedColumn[V, U]): Dataset[(K, U)]
	Computes the given aggregation, returning a Dataset of tuples for each unique key and the result of computing this aggregation over all elements in the group.

Using the General agg Operation

Just like on DataFrames, there exists a general aggregation operation `agg` defined on `KeyValueGroupedDataset`.

```
agg[U](col: TypedColumn[V, U]): Dataset[(K, U)]
```

The only thing a bit peculiar about this operation is its argument. What do we pass to it?

Using the General agg Operation

Just like on DataFrames, there exists a general aggregation operation `agg` defined on `KeyValueGroupedDataset`.

```
agg[U](col: TypedColumn[V, U]): Dataset[(K, U)]
```

The only thing a bit peculiar about this operation is its argument. What do we pass to it?

Typically, we simply select one of these operations from function, such as `avg`, choose a column for `avg` to be computed on, and we pass it to `agg`.

```
someDS.agg(avg($"column"))
```



Using the General agg Operation

Just like on DataFrames, there exists a general aggregation operation `agg` defined on `KeyValueGroupedDataset`.

```
agg[U](col: TypedColumn[V, U]): Dataset[(K, U)]
```

Typically, we simply select one of these operations from function, such as `avg`, choose a column for `avg` to be computed on, and we pass it to `agg`.

```
someDS.agg(avg($"column"))
// [error]  found    : org.apache.spark.sql.Column
// [error]  required: org.apache.spark.sql.TypedColumn[Listing,?]
// [error]                      .agg(avg($"column"))
// [error]                                ^
// [error] one error found
```

Oops. `TypedColumn`! Remember that we have to use `as[...]` to convert our untyped regular Column into a `TypedColumn`.

Using the General agg Operation

Just like on DataFrames, there exists a general aggregation operation `agg` defined on `KeyValueGroupedDataset`.

```
agg[U](col: TypedColumn[V, U]): Dataset[(K, U)]
```

Typically, we simply select one of these operations from function, such as `avg`, choose a column for `avg` to be computed on, and we pass it to `agg`.

```
someDS.agg(avg($"column")4.as[Double])
```

All better now.

Some KeyValueGroupedDataset (Aggregation) Operations

mapGroups

mapGroups[U](f: (K, Iterator[V]) => U): Dataset[U]

Applies the given function to each group of data. For each unique group, the function will be passed the group key and an iterator that contains all of the elements in the group. The function can return an element of arbitrary type which will be returned as a new Dataset.

flatMapGroups

flatMapGroups[U](f: (K, Iterator[V]) => TraversableOnce[U]): Dataset[U]

Applies the given function to each group of data. For each unique group, the function will be passed the group key and an iterator that contains all of the elements in the group. The function can return an iterator containing elements of an arbitrary type which will be returned as a new Dataset.

Note: at the time of writing, KeyValueGroupedDataset is marked as @Experimental and @Evolving. Therefore, expect this API to fluctuate—it's likely that new aggregation operations will be added and others could be changed.

reduceByKey?

If you glance around the Dataset API docs, you might notice that Datasets are missing an important transformation that we often used on RDDs: `reduceByKey`.

reduceByKey?

If you glance around the Dataset API docs, you might notice that Datasets are missing an important transformation that we often used on RDDs: `reduceByKey`.

Challenge:

Emulate the semantics of `reduceByKey` on a Dataset using Dataset operations presented so far. Assume we'd have the following data set:

```
val keyValues =  
  List((3,"Me"),(1,"Thi"),(2,"Se"),(3,"ssa"),(1,"sIsA"),(3,"ge:"),(3,"-"),(2,"cre"),(2,"t"))
```

Find a way to use Datasets to achieve the same result that you would get if you put this data into an RDD and called:

```
keyValuesRDD.reduceByKey(_+_)
```



Try it on your own now!

Note: the objective is just to use the APIs presented so far, don't worry about performance for now.

reduceByKey?

Challenge:

Emulate the semantics of reduceByKey on a Dataset using Dataset operations presented so far. Assume we'd have the following data set:

Dataset[(Int, String)]

```
val keyValues =  
  List((3, "Me"), (1, "Thi"), (2, "Se"), (3, "ssa"), (1, "sIsA"), (3, "ge:"), (3, "-")), (2, "cre"), (2, "t"))
```

```
val keyValuesDS = keyValues.toDS
```

- + -

```
keyValuesDS.groupByKey(p => p._1)  
  .mapGroups((k, vs) => (k, vs.foldLeft("")((acc, p) => acc + p._2)))
```

reduceByKey?

Challenge:

Emulate the semantics of reduceByKey on a Dataset using Dataset operations presented so far. Assume we'd have the following data set:

```
val keyValues =  
  List((3,"Me"),(1,"Thi"),(2,"Se"),(3,"ssa"),(1,"sIsA"),(3,"ge:"),(3,"-")),  
  (2,"cre"),(2,"t"))
```

```
val keyValuesDS = keyValues.toDS
```

```
keyValuesDS.groupByKey(p => p._1)  
  .mapGroups((k, vs) => (k, vs.foldLeft("")((acc, p) => acc + p._2))).show()
```

_1	_2
1	ThisIsA
3	Message:-)
2	Secret

reduceByKey?

Challenge:

Emulate the semantics of reduceByKey on a Dataset using Dataset operations presented so far. Assume we'd have the following data set:

```
val keyValues =  
  List((3,"Me"),(1,"Thi"),(2,"Se"),(3,"ssa"),(1,"sIsA"),(3,"ge:"),(3,"-)),(2,"cre"),(2,"t"))
```

```
val keyValuesDS = keyValues.toDS
```

```
keyValuesDS.groupByKey(p => p._1)  
  .mapGroups((k, vs) => (k, vs.foldLeft("")((acc, p) => acc + p._2))).show()
```

_1	_2
1	ThisIsA
3	Message:-)
2	Secret

Let's sort the records by id number! :-)

reduceByKey?

Challenge:

Emulate the semantics of reduceByKey on a Dataset using Dataset operations presented so far. Assume we'd have the following data set:

```
val keyValues =  
  List((3,"Me"),(1,"Thi"),(2,"Se"),(3,"ssa"),(1,"sIsA"),(3,"ge:"),(3,"-")),  
  (2,"cre"),(2,"t"))
```

```
val keyValuesDS = keyValues.toDS
```

```
keyValuesDS.groupByKey(p => p._1)  
  .mapGroups((k, vs) => (k, vs.foldLeft("")((acc, p) => acc + p._2)))  
  .sort($"_1").show()
```

_1	_2
1	ThisIsA
2	Secret
3	Message:-)

reduceByKey?

Challenge:

Emulate the semantics of reduceByKey on a Dataset using Dataset operations presented so far. Assume we'd have the following data set:

```
val keyValues =  
    List((3,"Me"),(1,"Thi"),(2,"Se"),(3,"ssa"),(1,"sIsA"),(3,"ge:"),(3,"-")),  
    (2,"cre"),(2,"t"))  
  
val keyValuesDS = keyValues.toDS  
  
keyValuesDS.groupByKey(p => p._1)  
    .mapGroups((k, vs) => (k, vs.foldLeft("")((acc, p) => acc + p._2)))
```

The only issue with this approach is this disclaimer in the API docs for mapGroups:

This function does not support partial aggregation, and as a result requires shuffling all the data in the Dataset. If an application intends to perform an aggregation over each key, it is best to use the ~~reduce~~ function or an org.apache.spark.sql.expressions#Aggregator.

reduceByKey?

Challenge:

Emulate the semantics of reduceByKey on a Dataset using Dataset operations presented so far. Assume we'd have the following data set:

```
val keyValues =  
    List((3,"Me"),(1,"Thi"),(2,"Se"),(3,"ssa"),(1,"sIsA"),(3,"ge:"),(3,"-")),  
    (2,"cre"),(2,"t"))  
  
val keyValuesDS = keyValues.toDS  
  
keyValuesDS.groupByKey(p => p._1)  
    .mapValues(p => p._2)  
    .reduceGroups((acc, str) => acc + str)
```

That works! But the docs also suggested an Aggregator?



Aggregators

A class that helps you generically aggregate data. Kind of like the aggregate method we saw on RDDs.

```
class Aggregator[-IN, BUF, OUT]
```

org.apache.spark.sql.expressions.Aggregator

- ▶ **IN** is the input type to the aggregator. When using an aggregator after groupByKey, this is the type that represents the value in the key/value pair.
- ▶ **BUF** is the intermediate type during aggregation.
- ▶ **OUT** is the type of the output of the aggregation.

Aggregators

A class that helps you generically aggregate data. Kind of like the aggregate method we saw on RDDs.

```
class Aggregator[-IN, BUF, OUT]
```

- ▶ **IN** is the input type to the aggregator. When using an aggregator after groupByKey, this is the type that represents the value in the key/value pair.
- ▶ **BUF** is the intermediate type during aggregation.
- ▶ **OUT** is the type of the output of the aggregation.

This is how implement our own Aggregator:

```
val myAgg = new Aggregator[IN, BUF, OUT] {  
    def zero: BUF = ... // The initial value.  
    def reduce(b: BUF, a: IN): BUF = ... // Add an element to the running total  
    def merge(b1: BUF, b2: BUF): BUF = ... // Merge intermediate values.  
    def finish(b: BUF): OUT = ... // Return the final result.  
}.toColumn
```

Emulating reduceByKey with an Aggregator

Let's return to our example of trying to emulate reduceByKey on a specific data set, and let's see if we can implement the aggregation part of our reduceByKey operation with an Aggregator.

```
val keyValues =  
  List((3,"Me"),(1,"Thi"),(2,"Se"),(3,"ssa"),(1,"sIsA"),(3,"ge:"),(3,"-"),(2,"cre"),(2,"t"))  
  
val keyValuesDS = keyValues.toDS
```

Emulating reduceByKey with an Aggregator

Let's return to our example of trying to emulate reduceByKey on a specific data set, and let's see if we can implement the aggregation part of our reduceByKey operation with an Aggregator.

```
val keyValues =  
  List((3,"Me"),(1,"Thi"),(2,"Se"),(3,"ssa"),(1,"sIsA"),(3,"ge:"),(3,"-"),(2,"cre"),(2,"t"))  
  
val keyValuesDS = keyValues.toDS  
  
val strConcat = new Aggregator[?, ?, ?]{ // Step 1: what should Aggregator's  
  def zero: ? = ???  
  def reduce(b: ?, a: ?): ? = ???  
  def merge(b1: ?, b2: ?): ? = ???  
  def finish(r: ?): ? = ???  
}.toColumn
```

(Int, String)

String

String

Emulating reduceByKey with an Aggregator

Let's return to our example of trying to emulate reduceByKey on a specific data set, and let's see if we can implement the aggregation part of our reduceByKey operation with an Aggregator.

```
val keyValues =  
  List((3,"Me"),(1,"Thi"),(2,"Se"),(3,"ssa"),(1,"sIsA"),(3,"ge:"),(3,"-"),(2,"cre"),(2,"t"))  
  
val keyValuesDS = keyValues.toDS  
  
val strConcat = new Aggregator[(Int, String), String, String]{  
  def zero: ? = ???  
  def reduce(b: ?, a: ?): ? = ???          // Step 2: what should the rest of  
  def merge(b1: ?, b2: ?): ? = ???          // types be?  
  def finish(r: ?): ? = ???  
}.toColumn
```

Emulating reduceByKey with an Aggregator

Let's return to our example of trying to emulate reduceByKey on a specific data set, and let's see if we can implement the aggregation part of our reduceByKey operation with an Aggregator.

```
val keyValues =  
  List((3,"Me"),(1,"Thi"),(2,"Se"),(3,"ssa"),(1,"sIsA"),(3,"ge:"),(3,"-"),(2,"cre"),(2,"t"))  
  
val keyValuesDS = keyValues.toDS  
  
val strConcat = new Aggregator[(Int, String), String, String]{  
  def zero: String = ???  
  def reduce(b: String, a: (Int, String)): String = ???          // Step 3: implement the  
  def merge(b1: String, b2: String): String = ???                  // methods!  
  def finish(r: String): String = ???  
}.toColumn
```

Emulating reduceByKey with an Aggregator

Let's return to our example of trying to emulate reduceByKey on a specific data set, and let's see if we can implement the aggregation part of our reduceByKey operation with an Aggregator.

```
val keyValues =  
  List((3,"Me"),(1,"Thi"),(2,"Se"),(3,"ssa"),(1,"sIsA"),(3,"ge:"),(3,"-")),  
  (2,"cre"),(2,"t"))  
  
val keyValuesDS = keyValues.toDS  
  
val strConcat = new Aggregator[(Int, String), String, String]{  
  def zero: String = ""  
  def reduce(b: String, a: (Int, String)): String = ??? // Step 3: implement the  
  def merge(b1: String, b2: String): String = ??? // methods!  
  def finish(r: String): String = ???  
}.toColumn
```

Emulating reduceByKey with an Aggregator

Let's return to our example of trying to emulate reduceByKey on a specific data set, and let's see if we can implement the aggregation part of our reduceByKey operation with an Aggregator.

```
val keyValues =  
  List((3,"Me"),(1,"Thi"),(2,"Se"),(3,"ssa"),(1,"sIsA"),(3,"ge:"),(3,"-"),(2,"cre"),(2,"t"))  
  
val keyValuesDS = keyValues.toDS  
  
val strConcat = new Aggregator[(Int, String), String, String]{  
  def zero: String = ""  
  def reduce(b: String, a: (Int, String)): String = b + a._2 // Step 3: implement the  
  def merge(b1: String, b2: String): String = ??? // methods!  
  def finish(r: String): String = ???  
}.toColumn
```

Emulating reduceByKey with an Aggregator

Let's return to our example of trying to emulate reduceByKey on a specific data set, and let's see if we can implement the aggregation part of our reduceByKey operation with an Aggregator.

```
val keyValues =  
  List((3,"Me"),(1,"Thi"),(2,"Se"),(3,"ssa"),(1,"sIsA"),(3,"ge:"),(3,"-"),(2,"cre"),(2,"t"))  
  
val keyValuesDS = keyValues.toDS  
  
val strConcat = new Aggregator[(Int, String), String, String]{  
  def zero: String = ""  
  def reduce(b: String, a: (Int, String)): String = b + a._2  
  def merge(b1: String, b2: String): String = b1 + b2  
  def finish(r: String): String = r  
}.toColumn
```

Emulating reduceByKey with an Aggregator

Let's return to our example of trying to emulate reduceByKey on a specific data set, and let's see if we can implement the aggregation part of our reduceByKey operation with an Aggregator.

```
val keyValues =  
  List((3,"Me"),(1,"Thi"),(2,"Se"),(3,"ssa"),(1,"sIsA"),(3,"ge:"),(3,"-"),(2,"cre"),(2,"t"))  
  
val keyValuesDS = keyValues.toDS  
  
val strConcat = new Aggregator[(Int, String), String, String]{  
  def zero: String = ""  
  def reduce(b: String, a: (Int, String)): String = b + a._2  
  def merge(b1: String, b2: String): String = b1 + b2  
  def finish(r: String): String = r  
}.toColumn // Step 4: pass it to your aggregator!  
  
keyValuesDS.groupByKey(pair => pair._1)  
  .agg(strConcat.as[String])
```

Emulating reduceByKey with an Aggregator

Let's return to our example of trying to emulate reduceByKey on a specific data set, and let's see if we can implement the aggregation part of our reduceByKey operation with an Aggregator.

```
val strConcat = new Aggregator[(Int, String), String, String]{
    def zero: String = ""
    def reduce(b: String, a: (Int, String)): String = b + a._2
    def merge(b1: String, b2: String): String = b1 + b2
    def finish(r: String): String = r
}.toColumn
```

```
keyValuesDS.groupByKey(pair => pair._1)
    .agg(strConcat.as[String])
```

```
[error] object creation impossible, since: it has 2 unimplemented members.
[error] the missing signatures are as follows.
[error]   def bufferEncoder: org.apache.spark.sql.Encoder[String] = ???
[error]   def outputEncoder: org.apache.spark.sql.Encoder[String] = ???
[error]   val strConcat = new Aggregator[(Int, String), String, String]{
[error]     ^
[error] one error found
```

Oops! We're missing 2 methods implementations. What's an Encoder?

Encoders

Encoders are what convert your data between JVM objects and Spark SQL's specialized internal (tabular) representation. **They're required by all Datasets!**

Encoders are highly specialized, optimized code generators that generate custom bytecode for serialization and deserialization of your data.

The serialized data is stored using Spark internal Tungsten binary format, allowing for operations on serialized data and improved memory utilization.

What sets them apart from regular Java or Kryo serialization:

- ▶ Limited to and optimal for primitives and case classes, Spark SQL data types, which are well-understood.
- ▶ **They contain schema information**, which makes these highly optimized code generators possible, and enables optimization based on the shape of the data. Since Spark understands the structure of data in Datasets, it can create a more optimal layout in memory when caching Datasets.
- ▶ Uses significantly less memory than Kryo/Java serialization
- ▶ >10x faster than Kryo serialization (Java serialization orders of magnitude slower)

Encoders

Encoders are what convert your data between JVM objects and Spark SQL's specialized internal representation. **They're required by all Datasets!**

Two ways to introduce encoders:

- ▶ **Automatically** (generally the case) via implicits from a `SparkSession`.
`import spark.implicits._`
- ▶ **Explicitly** via `org.apache.spark.sql.Encoder` which contains a large selection of methods for creating Encoders from Scala primitive types and Products.

Encoders

Encoders are what convert your data between JVM objects and Spark SQL's specialized internal representation. **They're required by all Datasets!**

Two ways to introduce encoders:

- ▶ **Automatically** (generally the case) via implicits from a `SparkSession`.
`import spark.implicits._`
- ▶ **Explicitly** via `org.apache.spark.sql.Encoder`, which contains a large selection of methods for creating Encoders from Scala primitive types and Products.

Some examples of ‘Encoder’ creation methods in ‘Encoders’:

- ▶ `INT/LONG/STRING` etc, for *nullable* primitives.
- ▶ `scalaInt/scalaLong/scalaByte` etc, for Scala’s primitives.
- ▶ `product/tuple` for Scala’s Product and tuple types.

Example: Explicitly creating Encoders.

```
Encoders.scalaInt // Encoder[Int]
```

```
Encoders.STRING // Encoder[String]
```

```
Encoders.product[Person] // Encoder[Person], where Person extends Product/is a case class
```

Emulating reduceByKey with an Aggregator

Let's return to our example of trying to emulate reduceByKey on a specific data set, and let's see if we can implement the aggregation part of our reduceByKey operation with an Aggregator.

```
val keyValues =  
  List((3,"Me"),(1,"Thi"),(2,"Se"),(3,"ssa"),(1,"sIsA"),(3,"ge:"),(3,"-"),(2,"cre"),(2,"t"))  
  
val keyValuesDS = keyValues.toDS  
  
val strConcat = new Aggregator[(Int, String), String, String]{  
  def zero: String = ""  
  def reduce(b: String, a: (Int, String)): String = b + a._2  
  def merge(b1: String, b2: String): String = b1 + b2  
  def finish(r: String): String = r  
  override def bufferEncoder: Encoder[BUF] = ??? // Step 4: Tell Spark which  
  override def outputEncoder: Encoder[OUT] = ??? // Encoders you need.  
}.toColumn  
  
keyValuesDS.groupByKey(pair => pair._1)  
  .agg(strConcat.as[String])
```

Emulating reduceByKey with an Aggregator

Let's return to our example of trying to emulate reduceByKey on a specific data set, and let's see if we can implement the aggregation part of our reduceByKey operation with an Aggregator.

```
val keyValues =  
  List((3,"Me"),(1,"Thi"),(2,"Se"),(3,"ssa"),(1,"sIsA"),(3,"ge:"),(3,"-"),(2,"cre"),(2,"t"))  
  
val keyValuesDS = keyValues.toDS  
  
val strConcat = new Aggregator[(Int, String), String, String]{  
  def zero: String = ""  
  def reduce(b: String, a: (Int, String)): String = b + a._2  
  def merge(b1: String, b2: String): String = b1 + b2  
  def finish(r: String): String = r  
  override def bufferEncoder: Encoder[String] = Encoders.STRING  
  override def outputEncoder: Encoder[String] = Encoders.STRING  
}.toColumn  
  
keyValuesDS.groupByKey(pair => pair._1)  
  .agg(strConcat.as[String])
```

Emulating reduceByKey with an Aggregator

Let's return to our example of trying to emulate reduceByKey on a specific data set, and let's see if we can implement the aggregation part of our reduceByKey operation with an Aggregator.

```
val keyValues =  
  List((3,"Me"),(1,"Thi"),(2,"Se"),(3,"ssa"),(1,"sIsA"),(3,"ge:"),(3,"-")),  
  (2,"cre"),(2,"t"))  
  
val keyValuesDS = keyValues.toDS  
  
val strConcat = new Aggregator[(Int, String), String, String]{  
  def zero: String = ""  
  def reduce(b: String, a: (Int, String)): String = b + a._2  
  def merge(b1: String, b2: String): String = b1 + b2  
  def finish(r: String): String = r // +-----+-----+  
  override def bufferEncoder: Encoder[String] = Encoders.STRING // |value|anon$1(scala.Tuple2)|  
  override def outputEncoder: Encoder[String] = Encoders.STRING // +-----+-----+  
}.toColumn // | 1| ThisIsA|  
keyValuesDS.groupByKey(pair => pair._1) // | 3| Message:-)|  
  .agg(strConcat.as[String]).show // | 2| Secret| // +-----+-----+
```

Common Dataset Actions

collect(): Array[T]

Returns an array that contains all of Rows in this Dataset.

count(): Long

Returns the number of rows in the Dataset.

first(): T/head(): T

Returns the first row in this Dataset.

foreach(f: T => Unit): Unit

Applies a function f to all rows.

reduce(f: (T, T) => T): T

Reduces the elements of this Dataset using the specified binary function.

show(): Unit

Displays the top 20 rows of Dataset in a tabular form.

take(n: Int): Array[T]

Returns the first n rows in the Dataset.

When to use Datasets vs DataFrames vs RDDs?

Use Datasets when...

- ▶ you have structured/semi-structured data
- ▶ you want typesafety
- ▶ you need to work with functional APIs
- ▶ you need good performance, but it doesn't have to be the best

Use DataFrames when...

- ▶ you have structured/semi-structured data
- ▶ you want the best possible performance, automatically optimized for you

Use RDDs when...

- ▶ you have unstructured data
- ▶ you need to fine-tune and manage low-level details of RDD computations
- ▶ you have complex data types that cannot be serialized with Encoders

Limitations of Datasets

Catalyst Can't Optimize All Operations

Take filtering as an example.

Relational filter operation E.g., `ds.filter($"city".as[String] === "Boston")`.

Performs best because you're explicitly telling Spark which columns/attributes and conditions are required in your filter operation. With information about the structure of the data and the structure of computations, Spark's optimizer knows it can access only the fields involved in the filter without having to instantiate the entire data type. Avoids data moving over the network.

Catalyst optimizes this case.

Functional filter operation E.g., `ds.filter(p => p.city == "Boston")`.

Same filter written with a function literal is opaque to Spark – it's impossible for Spark to introspect the lambda function. All Spark knows is that you need a (whole) record marshaled as a Scala object in order to return true or false, requiring Spark to do potentially a lot more work to meet that implicit requirement.

Catalyst cannot optimize this case.

Limitations of Datasets

Catalyst Can't Optimize All Operations

Takeaways:

- ▶ When using Datasets with higher-order functions like `map`, you miss out on many Catalyst optimizations.
- ▶ When using Datasets with relational operations like `select`, you get all of Catalyst's optimizations.
- ▶ Though not all operations on Datasets benefit from Catalyst's optimizations, Tungsten is still always running under the hood of Datasets, storing and organizing data in a highly optimized way, which can result in large speedups over RDDs.

Limitations of Datasets

Limited Data Types

If your data can't be expressed by case classes/Products and standard Spark SQL data types, it may be difficult to ensure that a Tungsten encoder exists for your data type.

E.g., you have an application which already uses some kind of complicated regular Scala class.

Limitations of Datasets

Limited Data Types

If your data can't be expressed by case classes/Products and standard Spark SQL data types, it may be difficult to ensure that a Tungsten encoder exists for your data type.

E.g., you have an application which already uses some kind of complicated regular Scala class.

Requires Semi-Structured/Structured Data

If your unstructured data cannot be reformulated to adhere to some kind of schema, it would be better to use RDDs.