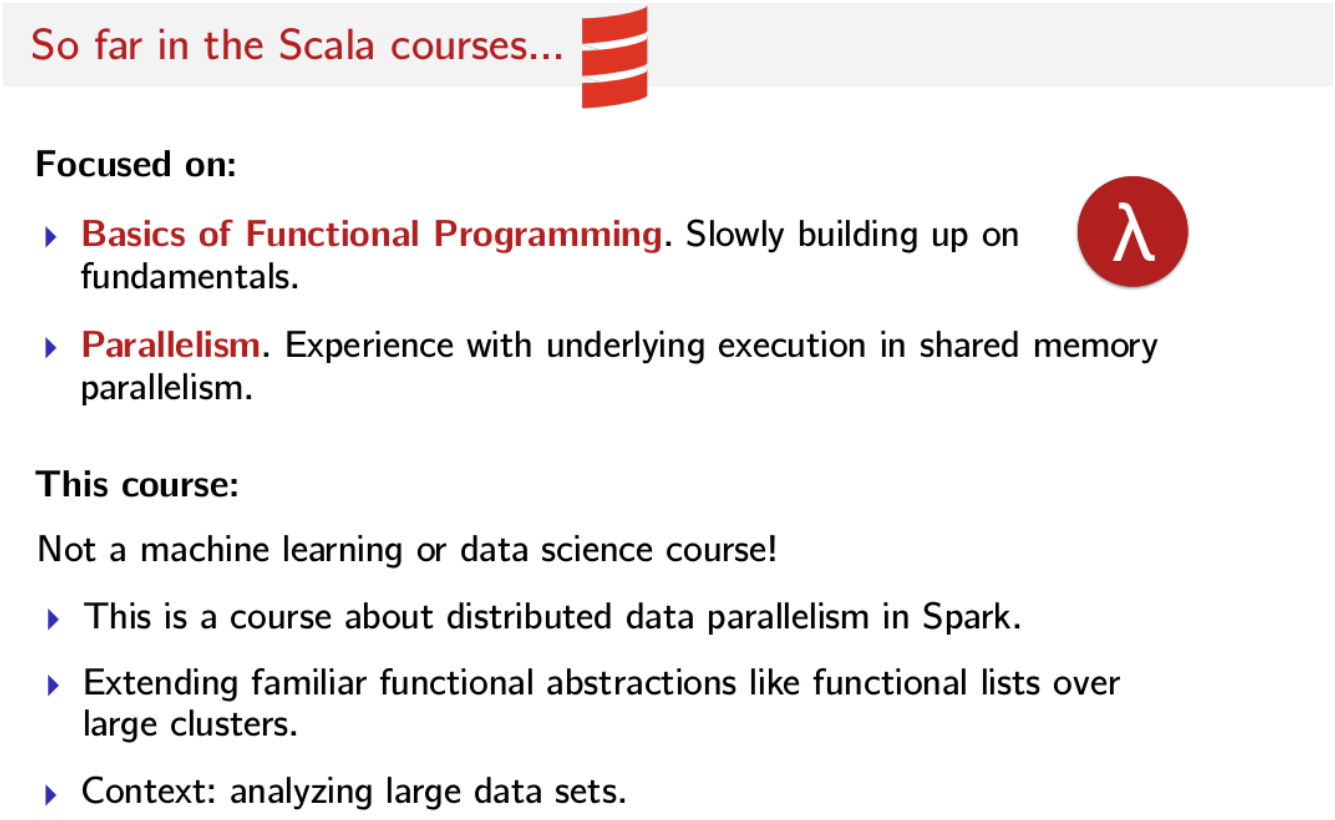
Introduction, Logistics, What You'll Learn



Hello and welcome to this course on Big Data Analysis in Scala and Spark. My name is Heather Miller, I'm your research scientist at EPFL, the executive director of the Scala Center at EPFL and an assistant clinical professor at Northeastern University in Boston. This course is all about taking some of the concepts you've picked up in earlier courses in the Scala specialization. So in particular, the functional programming in Scala course. And then applying some of these skills that you've learned to massive data sets using a popular framework written in Scala called Spark. So far throughout the series of Scala courses, we've focused on the basics of functional programming. And we focused on particular on sort of learning the fundamentals, and slowly but gradually building up more and more interesting programs from these fundamentals and this was in the courses number one and two. So principles of functional programming in Scala and functional programming design in Scala. Then we moved on to the Parallelism course where we started to focus on the underlying executions of our computations in a parallel setting. Now in this course, we're going to continue that trend. We're going to begin to think about applying some of these fundamental functional concepts across many machines rather than many processors. We'll also begin to shift our thinking towards a class of applications unlike we've done before. So we're really going to start looking at analyzing large amounts of data, which is typically the focus of data scientists. That said, it's important to note that this isn't the machine learning or data science course. Rather than focusing on machine learning algorithms or designing and tuning models, we will instead focus on how to map some of the functional abstractions that you've learned in previous Scala courses to computations on multiple machines over massive data sets. That is, we will see first-hand how the functional abstractions that we've covered in the previous Scala courses makes it easier and more user-friendly to scale computations over large clusters. Or easier, per se, than scaling computations on imperative frameworks, imperative systems for distributed computation. However as alluded to earlier in our program exercises, we're always going to focus on analyzing large data sets. That is you'll be challenged to think about common data science tasks like K-means functionally, such

as that they can be adopted to and implemented in the context of Spark. A functionally oriented framework for large scale data processing that's implemented in Scala.

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".

Before we go any further, you might be asking well, if we're going to be focusing on a lightweight data sciencey flavor of the processing tasks, then why are we bothering with Scala and why are we bothering with Spark? After all, if you want to learn data science in the classroom off of statistics professor's favorite languages or frameworks like R or Python or Octave and/or MATLAB. So then why should one bother running Scala or Spark which are both arguably very unlike R, Python, Octave and MATLAB? The answer is that these language and frameworks are good for data science in the small. Algorithms on data sets that are perhaps just a few hundred megabytes or even a few gigabytes in size. However, once the dataset becomes too large to fit into main memory on one computer, it suddenly becomes much more difficult to use one of these languages or frameworks alone. In short, if your small dataset grows into a much larger data set than these languages and frameworks like R, Python, MATLAB, etc. They won't allow you to scale, you'll need to start completely from scratch reimplementing all of your algorithms using a system like Hadoop or Spark anyway. We'll need to manually figure out how to distribute your problem over many machines without the help of such a framework. Which is kind of a bad idea if you're not already an expert in building distributed systems. And, yeah, there's also this whole huge massive industry shift towards data-oriented decision making. Nowadays, many companies across many different industries have realized that by looking more closely at the data they're collecting from device logs to health or genetic data, they can innovate in ways that were impossible before. For example, now we have all of these devices surrounding us, collecting information and attempting to provide all kinds of insights to enrich our day-to-day lives. Or instead, imagine hundreds of thousands of users of some device, say a smartphone or some wearable or something. And imagine as part of your job, you're responsible for providing some analysis or insight behind all of the data that's collected. They're providing insights to the smartphone's manufacturer about how the smartphone is operating. It would be nice, for example, if your smartphone manufacturer was able to catch an updated glitch before it became a big problem for you. Or analysis of your level of physical activity relative to the average activity of other users using the same wearable as you for example. In both of these cases, the language or the framework that you might've learned in a statistics class couldn't be used as you learned it. These are data science problems in the large that I'm talking about, and they can't be solved in a single compute node alone. On all of these kinds of problems, scale

far beyond the tech industry alone, you'll find similar problems in medical research. Lately, there's a number of initiatives focused on developing personalized treatments for disease for example. You'll also find similar problems in finance, manufacturing and many other areas of industry. In short, almost every industry has moved towards improving their business or products using some kind of data science, very often using 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 and feels a lot like Scala Collections!

So then let me ask you again, why Scala and why Spark? So okay, we established R and MATLAB, as you learn it in schools, isn't going to work for these data science in the large situations that have increasing importance across industries. But by using the language like Scala, it's easier to scale your problem to the large with Spark whose API is almost one-to-one with Scala's collections. That is by working with Scala in a functional style, you can quickly scale your problem out from one to tens, hundreds, or even thousands of nodes by leveraging Spark. A successful in performing large-scale data processing framework that looks a lot like Scala Collections.

Why Spark?

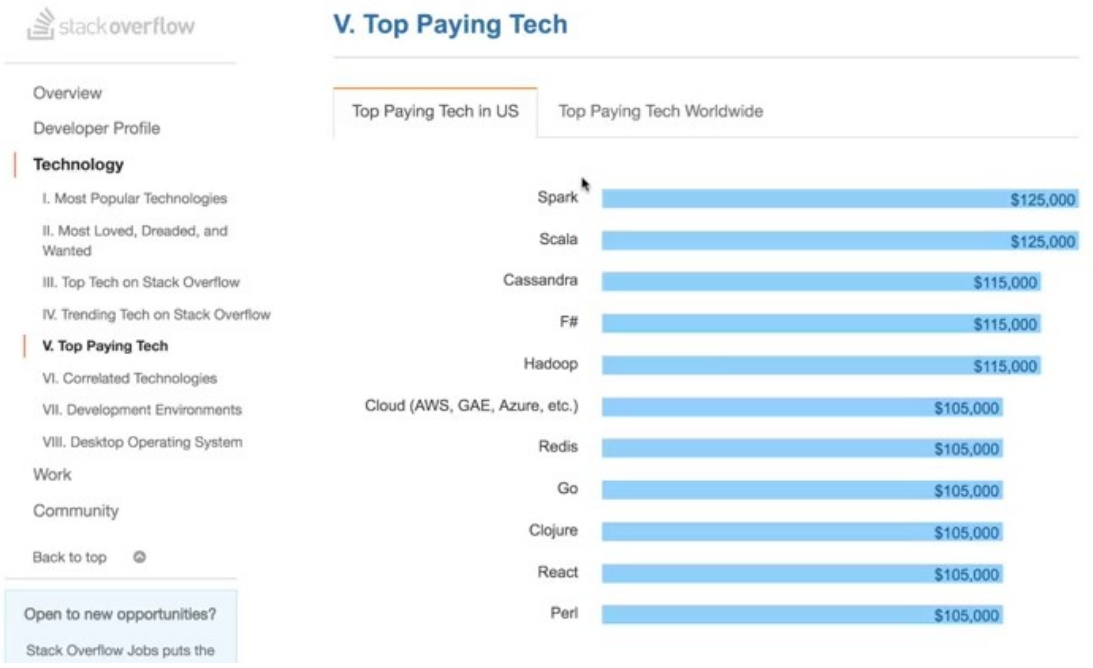
Spark is...

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

So let's start by touching on a few reasons to learn Spark. When it comes to dealing with large data sets, Hadoop is also a popular choice, so why would anybody bother with Spark? Well, there are few very strong reasons. One, Spark is more expressive. Spark's APIs are modeled after Scala's collections, which mean distributed computations in Spark are like immutable lists in Scala. You can use higher-order functions like map, flatMap, filter, and reduce, to build up rich pipelines of computation that are distributed in a very concise way. Whereas Hadoop on the other hand is much more rigid. It forces map then reduce computations without all of these cool combinators, like flatMap and filter, and it requires a lot more boilerplate to build up interesting computation pipelines. The second reason is performance. By now, I'm sure you've heard of Spark as being super fast. After all, Spark's tagline is Lightning-Fast Cluster Computing. Performance brings something very important to the table that we haven't had until Spark came along which is interactivity. Now it's possible to query your very large distributed data set interactively. So that's a really big deal. And also Spark is so much faster that Hadoop in some cases that jobs that would take tens of minutes to run, now only take a few seconds. This allows data scientists to interactively explore and experiment with their data, which in turn allows for data scientists to discover richer insights from their data faster. So something huge, which I feel isn't off dimension when people are talking about the pluses of Spark is that it really, really improves developer productivity. This is a really big point here, Spark improves the productivity of data analysts. And finally, Spark is good for data science. In fact, it's much better for data science than Hadoop and it's not just due to performance reasons. Iteration is required by most algorithms in the data scientist's toolbox. That is, most analysis tasks require multiple passes over the same set of data. And while iteration is indeed possible in Hadoop with really quite a lot of effort, you have a bunch of boilerplate that you have to do and a required external libraries and frameworks that just degenerate a bunch of extra map reduce phases in order to simulate iteration. It's really, on the other hand, the downright simple to do in Spark. There's no boilerplate required whatsoever, you just write what feels like a normal program, which includes a few passes over the same dataset. You just have basically, something that looks like a for loop and say hey, until this condition is met iterate. Which is night and day when compared with Hadoop, because it's almost not possible in Hadoop.

Also good to know...

Spark and Scala skills are in extremely high demand!



Another anecdotal reason that Spark and Scala together are both super interesting to learn at the moment is that Spark and Scala skills are in extremely high demand. If you don't know about it, there's actually this really cool developer survey that Stack Overflow does every year. And the 2016 survey results came out a few months ago and something really cool happened in these results. Let's just go to the technology section of the survey and look at the top paying tech. And as we can see, the top paying tech in the US is actually both Spark and Scala together. So indeed, Spark and Scala skills are both in high demand. And it's good thing you've already taken the Functional Programming Principles in Scala course, and now you're taking this Spark course?

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**

So what are we going to learn in this course? Well, we'll see how the data parallel paradigm that we learned in the parallel programming course can be extended to the distributed case using Spark. So that's where we're going to start in some of the next lectures. Then we're going to cover Spark's programming model in depth. We're then going to go into distributing computation, how to do it and how the cluster is actually laid out in Spark? We're then going to move on and spend quite a bit of time learning about how to improve performance in Spark. Looking at stuff like data locality and how to avoid recomputation and especially data shuffles in Spark. We're going to spend quite a bit of time trying to understand when data shuffles will occur. And finally, we're going to spend quite a bit of time diving into the relational operations available in Spark SQL module. To learn how to use the relational operations on DataFrames and Datasets and also all of the benefits that they bring you, as well as a handful of limitations. So Spark is popular, because it allows programmers to take advantage of massive resources from a handful to hundreds or even thousands of compute nodes. And it allows programmers to do so in a way where writing distributed programs feels like writing regular programs. As we'll see throughout this course, this detail, writing and programs that are distributed but with field sequential is very powerful. But it can also be the source of many headaches as we'll discover. Assumptions that you've learned to make as a programmer while working on single machines, have no longer hold on a cluster. And Spark does its best to make all kinds of smart decisions for us in order to ensure that our jobs run correctly all the way through. But as we'll see, we'll often need to understand a bit more about what's actually happening under the hood in Spark to get good performance out of it. So that's going to be a major theme of this entire course.

Prerequisites

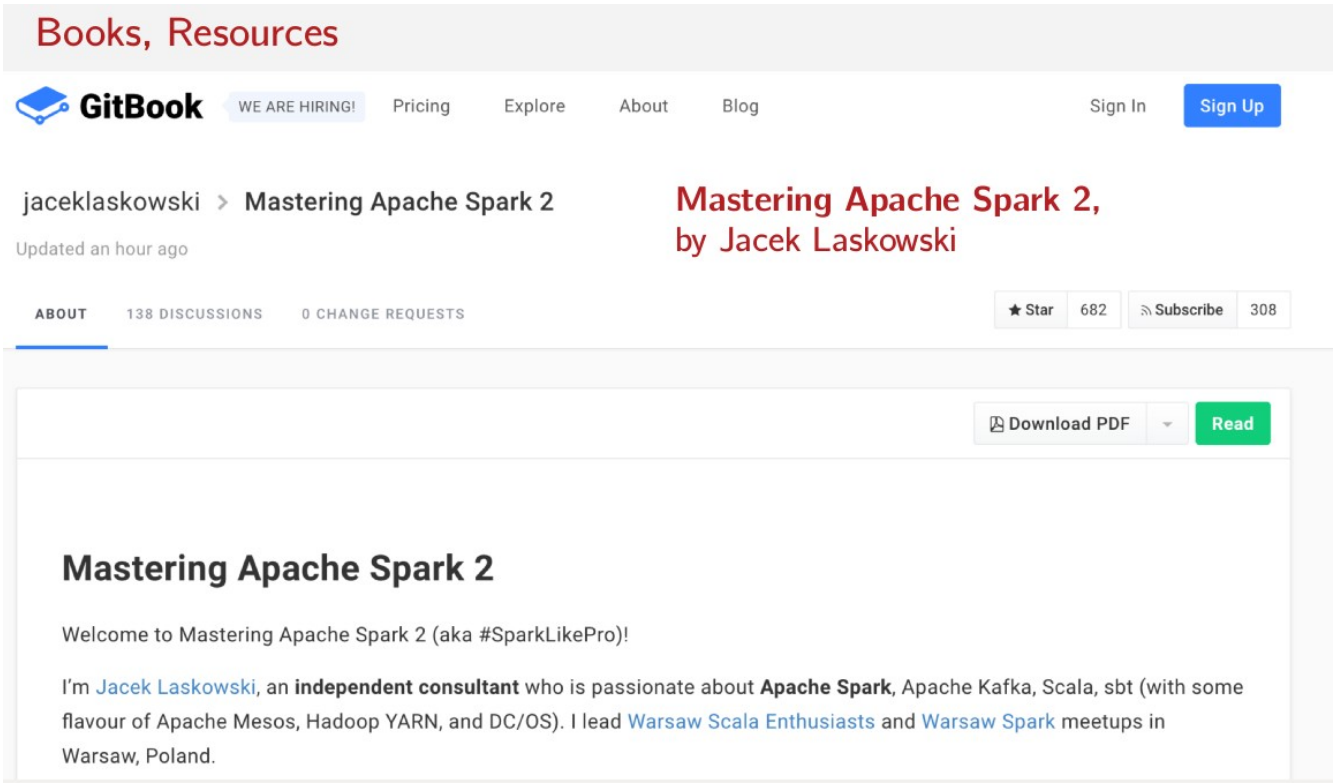
Builds on the material taught in the previous Scala courses.

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

Or at minimum, some familiarity with Scala.

So what are the prerequisites for this course? Well, it builds on the material taught in the previous Scala courses. So it would really be best if you've already taken the previous three Scala courses in this series. Those three courses are Principles of Functional Programming in Scala, Functional Program Design in Scala, and Parallel Programming in Scala. Though at minimum, some familiarity with Scala is required, whereas it's actually designed to connect to the Parallel Programming course that Viktor Kuncak and Alex Prokopetz taught. So many of the concepts that were introduced there, I pick up on them and I extend those concepts to the distributed case. So its also helpful to at least have some familiarity with sort of the concepts that we're taught in the parallel programming course. What about books and other resources that could help you out in this course? Well, in the past year or so several books covering Spark have emerged and most of them teach Spark in Scala. A good book covering the basics is O'Reilly's Learning Spark written by Holden Karau, Andy Konwinski, Patrick Wendell, and Matei Zaharia. Matei's now a professor who created Spark when he was a graduate school at Berkeley. Another book that does a good job at covering the basics of Spark is Spark in Action which was published in 2017. It's full of examples and it's another good resource for sort of getting into Spark for the first time if you're looking for a good book. Go for a book that goes in to more detail on how to achieve good performance. The O'Reilly book called High Performance Spark which is currently in develop by Holden Karau and Rachel Warren is an excellent resource. This book goes into more depth about, precisely, how Spark executes jobs and how one can use that knowledge to squeeze better performance out of your Spark jobs. Then we'll cover in this course. So if you're looking for more performance, that's a great book to pick up. And finally, for a more data science focused look. At Spark, O'Reilly's Advanced Analytics with Spark book, written by Sandy Ryza, Uri Laserson, Sean Owen, and Josh Wills is a good book to pick up. This book is full of example data science applications implemented in a functional style with the Spark, and many of its major modules. So modules like live graphics, which we're not going to cover in this course. Further, the authors don't just stop at implementing the core algorithms. They go into detail about how to prepare and clean data, and how to tune models in order to get good results. So it's really a good resource for general data science as well. I highly recommend this book if you're interested in mapping some of the things you learned in this class to concepts and algorithms from machine learning. And finally if really, really zooming in on Spark is what you're after. There's this book published on GitBook called Mastering Apache Spark 2 by Jacek Laskowski, it covers Spark internals in great detail and it's constantly updated. It's a work in progress, but it's full of very cool information and all kinds of examples. So if you're really into the nitty-gritty,

have a look at this book.



As far as tooling goes, there are a couple of tutorials that you're going to find. As far as to tooling goes, there are a couple of few tutorials that you're going to find. As far as tooling goes, there's a couple of tutorials that you're going to find in the getting started section on the first week. If you've taken the other courses, there's nothing new here. The only required tools for this course are some IDE or text editor of your choice and sbt. There's another optional tool which you may be interested in using called Databricks Community Edition. Which is a hosted version of Spark that has some in-browser notebook that lets you interact with this hosted version of Spark. So that means, you wouldn't have to setup Spark anywhere, you could just go to a website and play with it. Here's just a quick glimpse of this Databricks Community Edition that I was telling you about. So you can get access to this platform for free by going to the Databricks website. Once you log in, this is what you have in front of you. But here, you have a dashboard and you could open notebooks and create your own notebooks. But there's all kinds of cool notebooks that have lots of example programs and of course, they let you run your own program. So this is a really cool platform that you may find useful while learning. Like all of the other Scala courses, this course comes with autograders. There are tutorials in the getting started section of the first week to teach you show to submit your solutions to our automatic graders and how to check your results. And this course features three auto-graded assignments that require you to do analyses on real-life data sets. So we didn't give you fake datasets, we gave you real datasets. So you can actually play around with them beyond but we ask you to do as part of your assignments in the course and there's all kinds of interesting insights to find. And that's all I have to say about logistics. So let's just dive in and get our hands dirty with Apache Spark and Scala.

Data-Parallel to Distributed Data-Parallel

ÉCOLE POLYTECHNIQUE
FÉDÉRALE DE LAUSANNE

Data-Parallel to Distributed Data-Parallel

Big Data Analysis with Scala and Spark

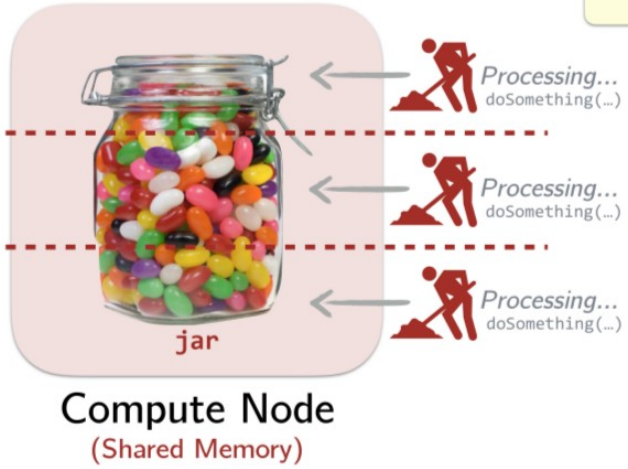
Heather Miller

In this session, we're going to try and bridge the gap between data parallelism in the shared memory case. Which is what we'd learned in the parallel programming course and distributed data parallelism. So taking that idea of data parallelism and extending that to the situation where you no longer have data on just one node anymore. Now you have data spread across several independent nodes.

Visualizing **Shared Memory** Data Parallelism

What does data-parallel look like?

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



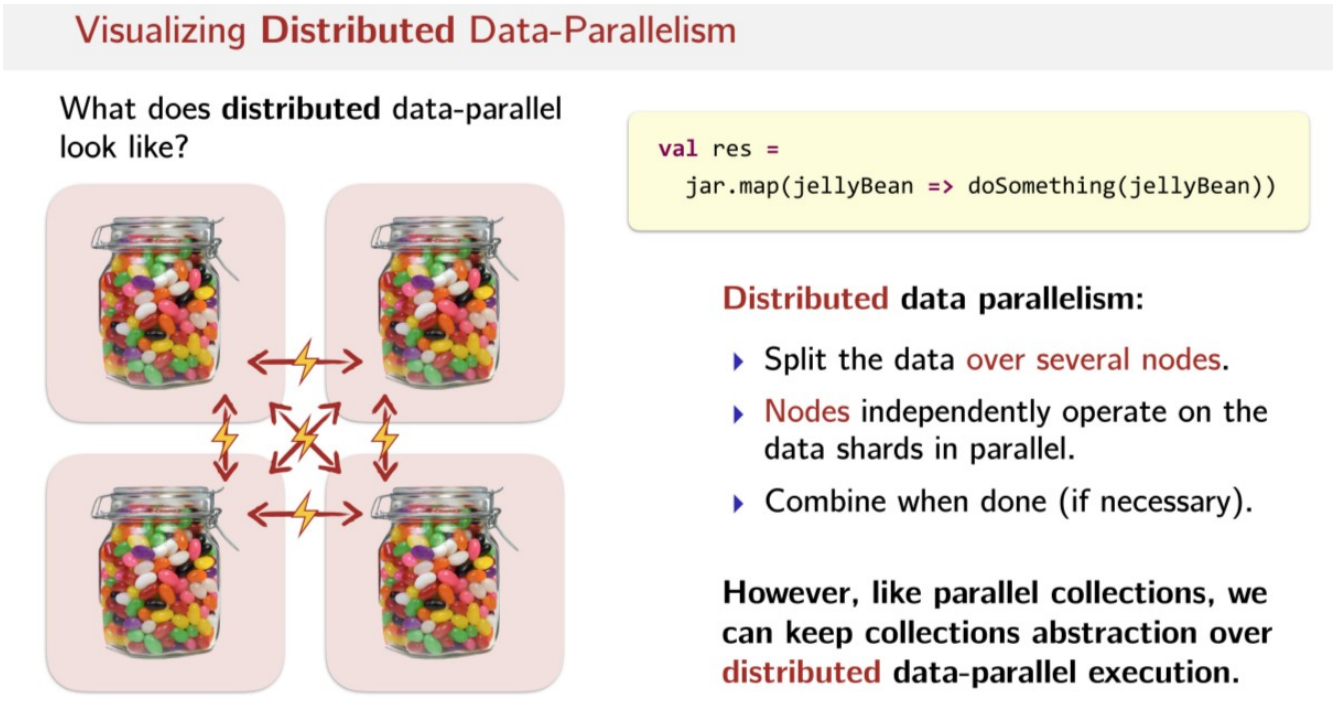
Shared memory data parallelism:

- 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.

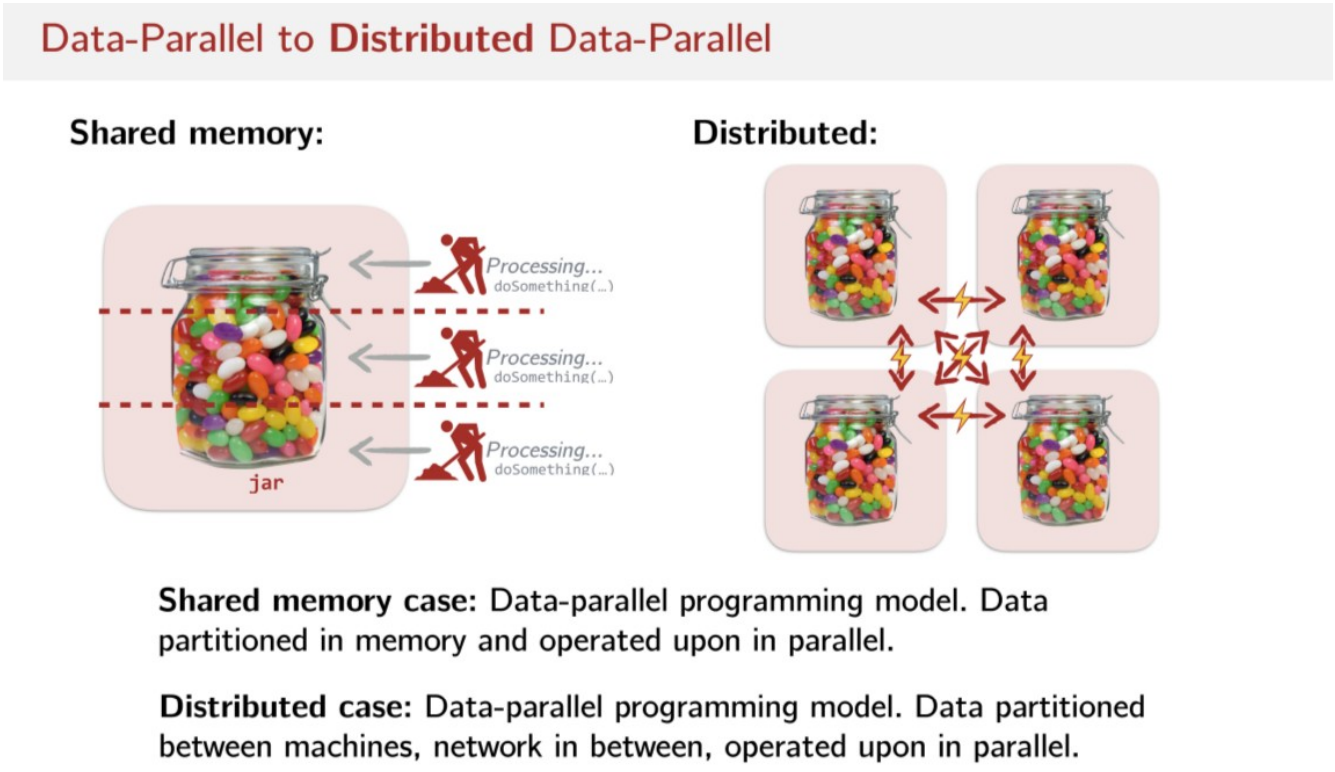
As usual let's get started by trying to develop a little bit of intuition first. Let's try to visualize this differences between shared memory and distributed data parallelism. So let's start with the short memory case, so if I had to draw a picture of data parallelism, what does it look like? Since we're talking about the shared memory case, we assume just one node, one computer. That means we have some dataset on one computer. And we'd like to exploit the ability of the computers multiple processors

to try and compute this data more quickly by doing it in parallel, by breaking up the work and trying to do many things at once. So let's assume that our data is in a collection, it's got a lot of parallel collection. So this jar is going to be a collection of jelly beans, let's just say. And our collection of jelly beans is called jar. It doesn't matter if we're doing this in parallel collections or sequential collections. We generally have the same API available to us. We can say, okay, I have this collection in a jar and I can do a map on that collection. So I want to do the same operation to all of the jelly beans in the jar. So for each jellybean in the collection, I'm going to do something to it. I don't know whatever it is, maybe I change the color. Now in the shared memory case, you can have this API. But under the hood, what parallel collections will do if we use parallel collections in Scala, is it will somehow split up the data. So there's some way to make chunks of the data. And then to have some kind of task or worker or some kind of thread abstraction, something that does processing on these individual data shards and parallels. So typically, you have many of these and whenever that work is done, we start combining the result into a complete result again. If necessary, right? So visually, we can chunk up our dataset into several pieces. And so we have something like a worker or a thread doing the individual work on individual pieces of that data all parallel at the same time. So again, this is the shared memory case, we have all of this data sitting in memory. And then different pieces of that data are being worked on by different individual processes, whatever our process is realized as, okay? So that's the visual picture of the shared memory case. What's important to note is that this can all live underneath a collections abstraction. So for example, we can take exactly the same abstraction that we already used in the case of regular collections. So for all you know because I've left out the types here. This jar could be a list or it could be a parallel array, you don't know. It has the same API, looks exactly the same as a regular Scala collection or a paralell collection. So the point here is we can have underneath the hood this data paralellism automatically happening for us. And we can reuse the same already very familiar and comfortable API that people already use. So key takeaway here is that that this collections abstraction fits over this data-parallel model very well.



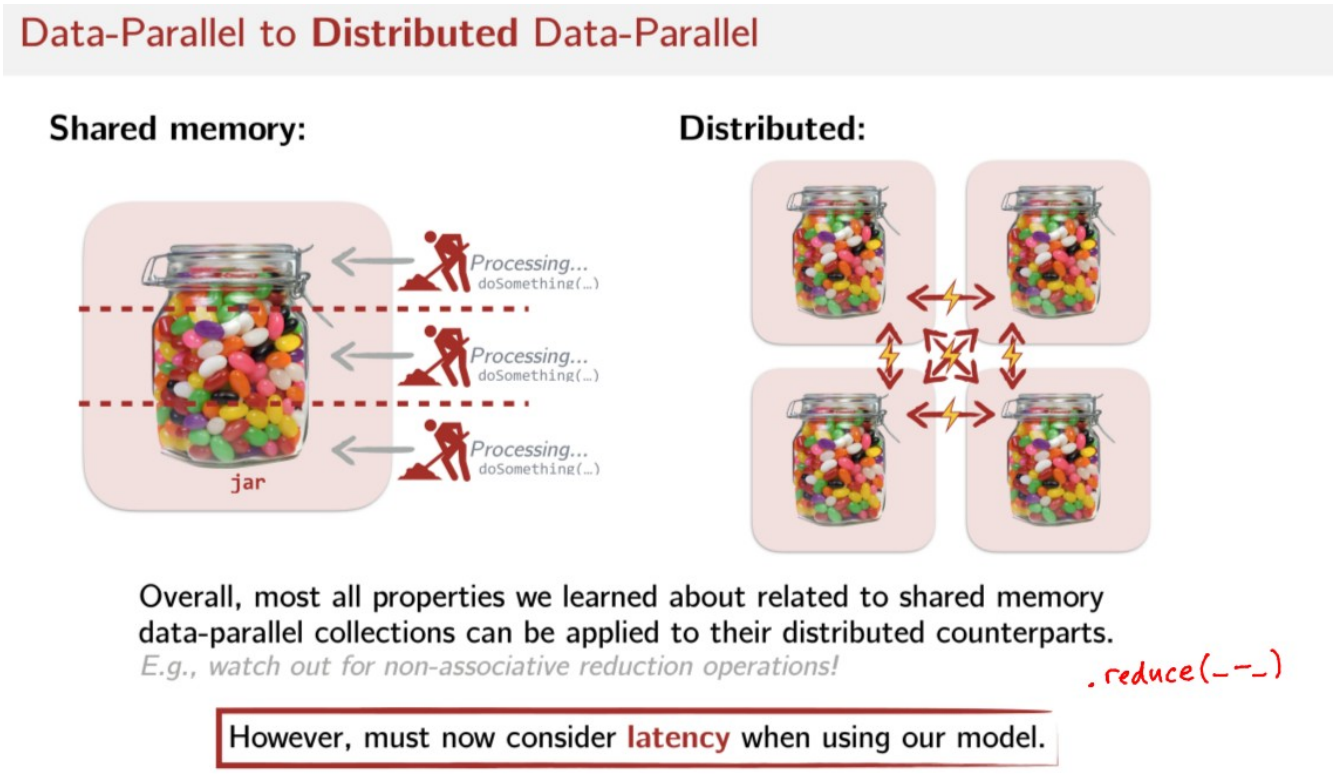
And we can extend this to the distributed case as well. So what would the same data apparel of execution look like in the distributed case sort of under the cover independent of the abstraction? So in the case of distributed data parallelism, as we saw in the shared memory data parallelism, we split the

data on the same machine, in the distributed case, we split the data over several nodes. Okay, that's the difference, we split it up over several nodes. In the shared memory case, workers or threads independently operate on the data shards in parallel. In the distributed case, nodes operate on the shards in parallel. So it's not workers or threads, it's independent nodes, independent machines working on the data in parallel, okay? So this very high level description of data parallelism in the distributed case is very similar to the shared memory case. The only difference really is that we're splitting things over several nodes. And the nodes are the things that do the work, instead of individual threats. So what does this look like? We can go back to the example of having some kind of collection. Once again lets have a collection of jelly beans. In this case, what we have done is we got individual nodes,, individual compute nodes. And each one has a piece of a dataset as an independent collection on each independent node. But one thing that we didn't have to deal with in the parallel programming course. The prerequisite to this course, when we were learning about data-parallelisms in the shared memory case, was the network, was the latency involved with these nodes having to share data or to communicate with one another in some way. The latency imposed by needing to do this, was never a worry when we were learning about shared memory data parallelism. So this is a big, fundamental difference now that we'll see in later lectures is actually going to impact the programming model. The latency is something that we will never be able to forget about. However, just like in parallel collections, we can still keep the same familiar collections abstraction over our distributed data-parallel execution. So just like before, we can have code that looks just like this. And in this case, a jar can be one of these distributed collections and spark. And yet we have this wonderful collections API that looks just like Scala collections, over top of this distributed data-parallel execution. So this model in general, fits very, very nicely underneath this collections abstraction is sort of the key takeaway here.



So just to summarize the differences between shared memory and distributed data-parallels. In the shared memory case, you have this data-parallel programming model, this collections programming model. And underneath the hood, the way that it's actually executed is that the data is partitioned in memory. And then operated upon in parallel by independent threads or using a thread pool or

something like that. Yet in the distributed case, we have the same collection abstraction we did in parallel model on top of this distributed execution. But now instead we have data between machines, the network in between which is important. And just like in a shared memory case we still operate on that data in parallel.



So what does this mean? It means that a lot of what we learned in the previous course, in the shared memory data parallelism part of that course, can actually be applied to the distributed counterparts. For example, non associative abduction operations. So just to remind you what that is. If you had a parallel collection and you did reduce (.reduce(_-_)) on it. And you subtracted the elements in your reduction operation, that's a non associative operation. And in parallel collections, that will give you a non deterministic result. Spark is exactly the same way, when you do these things in parallel. And they're non-associative when you have the non-associative reduction, the same exact thing happens, you could have non deterministic results. So the same things that we learned, the same sort of key take away. So that we got from the short memory data parallelism part of the parallel programming course. We can apply a lot of that the distributed data parallelism in Spark. However, the one thing we have to really think about carefully when we're using Spark's program model is to think about latency. And we'll see that more in subsequent lectures.

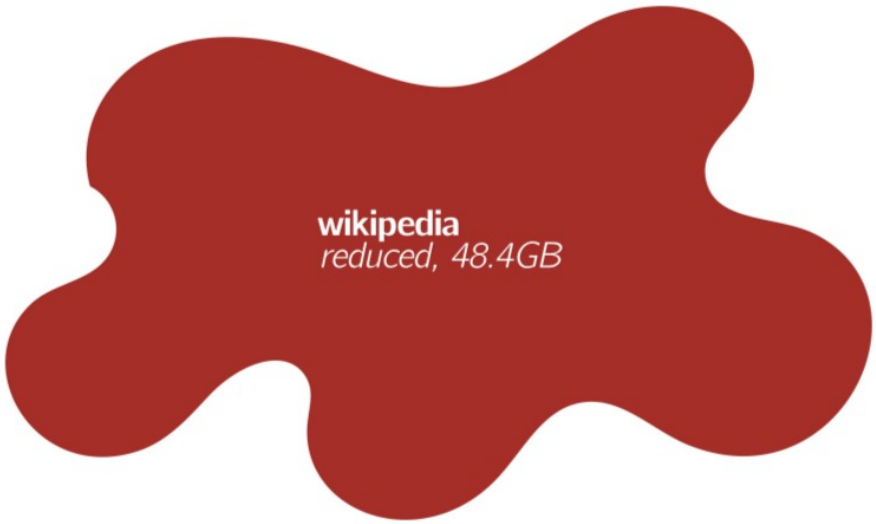
Apache Spark

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)

So as you know the implementation of this distributed data-parallel program model is Apache Spark. Apache Spark has an abstraction called resilient distributed datasets. We call them RDDs for short, and an RDD is basically the distributed counterpart of a parallel collection. So throughout the course we're going to be focussing a lot on these RDDs. How they work and how to get the best performance out of them.

Distributed Data-Parallel: High Level Illustration

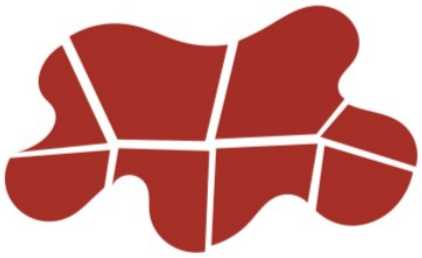


Given some large dataset that can't fit into memory on a single node...

This is a last sort of high level illustration that I'm going to give you about sort of how to think about Spark and how it works. So let's assume that we have a very large dataset, let's assume for example we have the english Wikipedia which we've reduced. So we don't have all of the possible data per article. We have just the titles and the texts of those articles, and just the english part of Wikipedia. And already that reduce sentence is pretty big, we can put that the memory of one machine unless you have really special machine. But most normal people has laptops can't put this into memory. So we know we have

to split it up over a couple of notes.

Distributed Data-Parallel: High Level Illustration



Distribute it over your cluster of machines.

So assuming we have a dataset like this, what Spark will do is it will chunk up this dataset somehow via some partitioning mechanism And then it will distribute this partitioned dataset over a cluster of nodes.

Distributed Data-Parallel: High Level Illustration

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

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

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

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

And what you get back from Spark is a reference to this distributed dataset, this entire distributed datasets. So now you have a very large collection that's spread over in this example, let's say eight

machines. And you have one name to refer to all of that distributed data with. And now I can think of it as if it's a single collection, because to me it's going to look like a single collection abstractions like a single list or something, right? So I can now refer to this thing called Wiki. And in fact, I'm referring to the distributed data that's distributed all over these, for example eight machines here, okay? I can then just use this collection abstraction that we're already very comfortable with. So this thing we've named it Wiki, and now I can just do a map on this Wiki for every single article. I can adjust the text, I can get text, and I can convert the text to all lowercase. So getting rid of any uppercase characters in the Wikipedia text, for example. This is just a high level picture of how we're going to be interacting with this distributed data. And as you can see, it's going to look very familiar. It's going to look just like collections that you're already very comfortable with except that it's going to be distributed over many nodes.

Latency

So before we get into how to use Spark, how to get good performance out of Spark, and how to express basic analytics jobs in Spark's programming model, let's first look at some of the key ideas behind Spark in an effort to get a bit of intuition about why Spark is causing such a shift in the world of data science and analytics. As we'll see in this section, Spark stands out in how it deals with latency, which is a fundamental concern when a system becomes distributed.

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.

So, earlier in the Sequence of courses and the Parallel Programming course, we learned about Data Parallelism in the single machine, multi-core, multiprocessor world. So, we thought about how to parallelize something in this data parallel paradigm, in the shared memory scenario. So, one machine. And we saw parallel collections as an example of an implementation of this paradigm. Today what we're going to do is we're going to extend what we've learned from the parallel programming course when we were focused on shared memory. We're going to extend that to the distributed setting. And as a real life implementation of this paradigm, we're going to get into Apache Spark, which is a framework for big data processing that sits in the Hadoop ecosystem.

Distribution

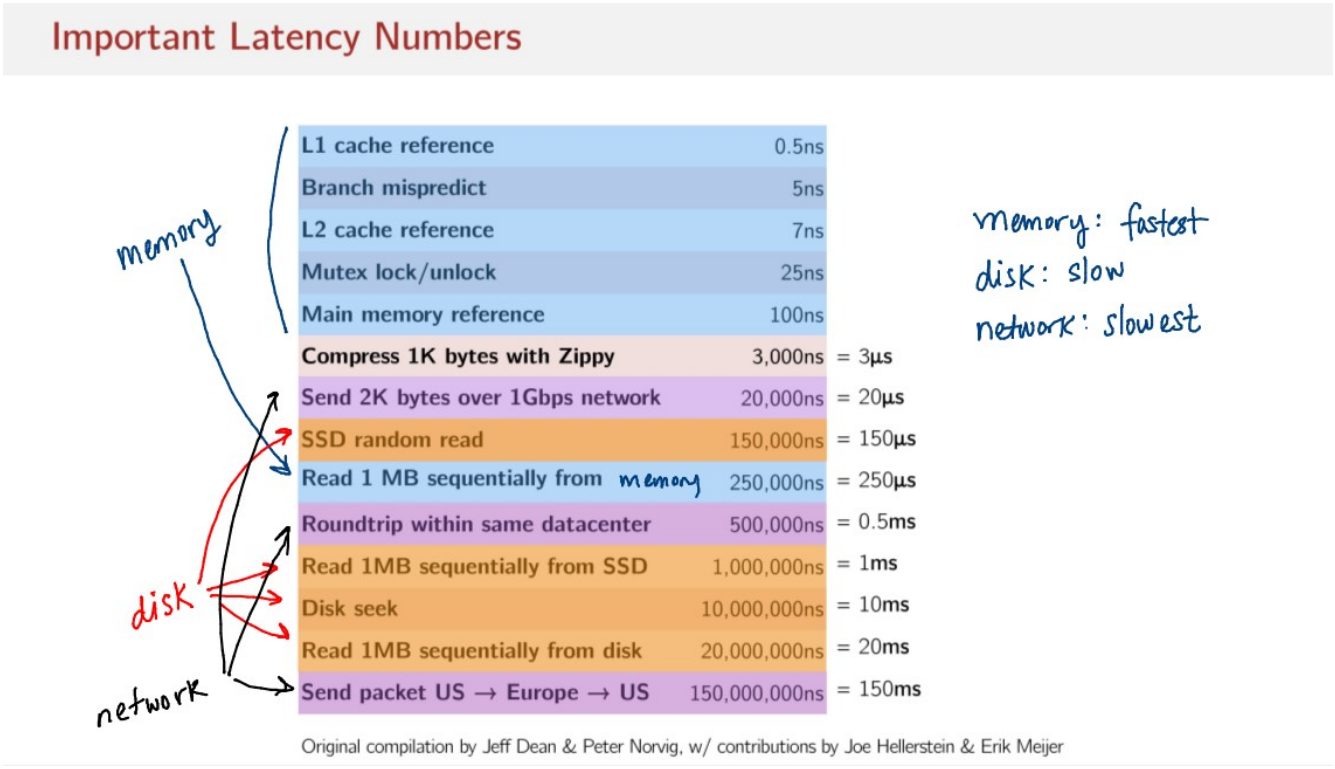
Distribution introduces important concerns beyond what we had to worry about when dealing with parallelism in the shared memory case:

- *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*.

So we've started with shared memory, one machine. And now, we're shifting to many machines, and many machines working together to complete a data parallel job. However, when we switch into this distributed paradigm, suddenly there are a lot more things that could go wrong that were not even of concern in the shared memory case. The list is actually much longer than these two points here. However, these two points are really the most important points to keep in mind when you're thinking

about Spark and, in general, other frameworks for distributed computing. In particular, Spark handles these two issues particularly well. So, one issue is partial failure, the situation where you might have one or a few machines involved in a large computation. For some reason, failing. Maybe an exception gets thrown, maybe the network between a master node and that worker node goes down. Something happens where suddenly one machine no longer is available or is no longer able to contribute its piece of the work. This is something that wasn't really an issue in the case of shared memory. Another issue is latency. There's this general observation that sometimes certain operations will have a much higher latency than other operations. So, in particular, network communication, we'll see, is very expensive. Which is something that is a reality in order to the any sort of distributed job. You cannot get rid of the need to communicate between nodes when you're doing a distributed computation. So, just to summarize, latency is a really big concern that pops up, that can really affect the speed at which a job is done. And partial failure can really affect whether or not a job can even be completed. What happens if a machine goes down and we have no way of recovering what that machine was working on? So, Spark handles these two issues particularly well. And the big take away that I hope you get from this lecture and some of the others in the series is that fundamental concerns to distribution, like latency, they cannot be masked completely, they cannot be forgotten about. You can't pretend that this does not exist. You cannot pretend that the network is not involved in the computations that you have to do. You always have to think about it. And, in fact, things like latency actually bubble up into the programming model. Perhaps it's not immediately obvious, but we'll see over the course of several lectures that dealing with latency actually gets into the code that you're writing, you're actually reasoning a little bit about whether or not you're going to cause network communication. So this is one very important thing to remember. When you do a distributed job, you're always going to have to remember that you're going to cause some kind of network communication. And because it's often expense, you typically want to reduce the amount of network communication that you cause.



Okay, so let's back up some of these loose arguments that I'm making. I'm saying, okay, there's a spectrum, and sometimes in memory computations are cheaper, computations that involve the network

are expensive, and perhaps writing things to disk and reading things from disk can also be expensive. And as a programmer, you should be aware of when you're doing some of these things. So there's a famous chart called Important Latency Numbers that every programmer should know. These numbers were originally compiled by Jeff Dean and Peter Norvig at Google, and also later extended by professor Joe Hellerstein and professor Erik Meier. And the idea here, is to actually show the relative amount of time that different sorts of operations cost. And okay, so what do we have here? We have a bunch of numbers, lots of zeros. Okay, I can see that this is cheaper and this is more expensive. So, this is half of a nanosecond, this is a million nanoseconds, 150 million nanoseconds. We can see that there are some pretty big orders of magnitude differences between some of these operations. Okay, that's fine. Okay, so let's zoom in on two of these numbers here. So, this is reading one megabyte sequentially from memory. There's a typo in the slides. And this is reading one megabyte sequentially from disk. Now, if we look at the numbers here, if we focus on, in particular, the nanoseconds, we can see that there is around approximately a 100 times difference between reading 1 MB sequentially from memory and reading 1 MB sequentially from disk. That's a pretty big difference. Likewise, if we look at the costs of referencing something that exists in main memory and sending a packet all the way from a data center in the US to a data center in Europe and then back to the data center in the US, there's likewise a very big difference here, too. So, again, if we focus on the nanoseconds, we can see that there's a 1 million times difference in the amount of time that it takes to reference something that's in memory or to do a round trip sending piece of data over the network and back. That means that it's 1,000,000x slower To send a packet round trip over over a long distance than it is to simply reference something that exists in main memory. That's a huge difference. Okay, so let's try to make a little sense of these numbers. So we can see that they're going from smallest to largest. And there's kind a little bit of a trend here where we can organize things into groups. So, the stuff that's blue refers to things that tend to happen in memory. They involve no disk, they involve no network. This as well. The things that are orange, they tend to involve disk. And finally, the things that are purple involve network. So, in this case, we're sending two kilobytes of data one way over a local area network. In this case, we're doing a round trip in the same data center in the same network. And in this case, we're doing another round trip over a very long distance. Things that involve network in purple. Okay, so this is perhaps getting to be a little bit confusing with all the arrows. But what we see is a general trend. Memory operations tend to be faster. Disk operations tend to be pretty slow, but not horribly slow. And any operation involving the network. Tends to be, the slowest. Okay, so we have this general feeling that doing things in memory is fast. Doing things on disk is kind of slow. And doing stuff that involves a lot of network communication is really slow. We have some numbers, we say. We can see that this versus this is 1 million times slower than that. And this here, reading 1 MB sequentially from memory, this is 100 times. Reading 1 MB sequentially from disk is 100 times slower than reading it from in-memory. So we have these numbers, 100 times slower, 1 million times slower. But, these are orders of magnitude and it's really difficult to wrap your mind around the cost of these things when it's just some orders of magnitude.

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*.

So let's try to humanize these numbers. Let's try to get a better intuition between really the meanings Of these order of magnitude differences between these latency numbers. So, to do that let's multiple all of these numbers that we just saw by one billion. So now, suddenly, instead of nanoseconds we're dealing with seconds. So we can then map each of these latency numbers. To human activity. So, we just kind of shift this into our realm of understanding, our realms of times that we operate on, and then look at how expensive some of these things are, relative to our lives.

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

So, let's have a look at some of these latency numbers match some kind of human activity. And I should note that these come from GitHub just posted by Joe Hellerstein. So, you can find these online on GitHub and search for, Humanized Latency Numbers. So, let's look at some of these numbers. So, we can start to group things into categories. Things that take seconds to a minute, things that take hours, things that take days so on and so forth. So if we start at the top of our list we have our nanoseconds now map to seconds. We have L1 cache reference. If it's equivalent to have a second of time, that's about one human heart beat. Okay so, we have some feeling for how fast referencing something in our

L1 cache Is. And then the L2 cache it's about 7 seconds comparatively so kind of the equivalent of a long yawn. Locking and unlocking things in memory about 25 seconds which is about the amount of time it takes to make a push button coffee, okay. So, we keep moving down our list referencing main memory is equivalent to about 100 seconds which. Depending on the person, is how long it takes to brush your teeth. Okay?

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

So, now we're starting to get into some of the first networking numbers. Sending two kilobytes of data one way over a one gigabyte network. That's about five and a half hours. Woah, that's a big difference already. So on a local area network, one way send is equivalent to about Five and a half hours relative to our half second l one cash reference. That's huge, so the difference goes from one heart beat to the nano time, that it takes for you to go to lunch and then leave your job at the end of the day. That's a pretty big difference, and so lets continue. A random read on the SSD 1.7 days about the duration of a normal weekend. This is getting crazy isn't it? So, we go back now to reading 1 megabyte sequentially from data that’s about 2.9 days even longer that like a weekend plus a holiday. Round trip within the same datacenter 5.8 days that kind of a one week vacation. Reading 1 MB sequentially from SSD, 11.6 days, that's almost two weeks. That's an enormous difference compared to an L1 cache reference. Again, half a second to 11.6 days, and as programmers we have a tendency to just Intermix these things without really having a good appreciation of these order of magnitude differences. We don't really have a good feeling for how different these things are.

More Humanized Latency Numbers

Year:

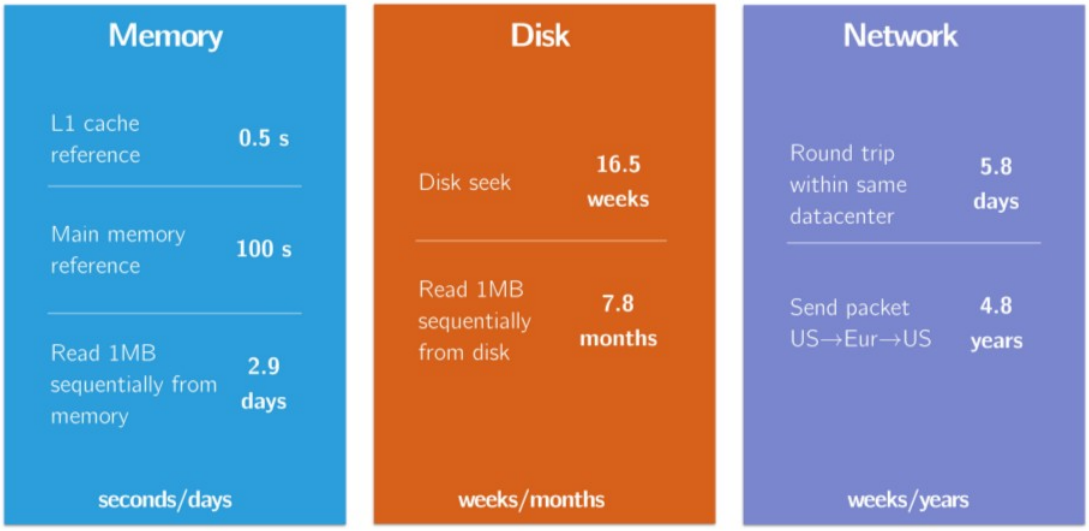
Disk seek	16.5 weeks	A semester in university
Read 1 MB sequentially from disk	7.8 months	Almost producing a **new** human being
The above 2 together	1 year	

Decade:

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

So we can keep on going reading 1 MB sequentially from disk is 7.8 months. That's almost as long as it takes to. Have a human pregnancy start and a baby be born. So, the difference between referencing L1 cache and reading 1 megabyte sequentially from disk is enormous. And finally, this example of a long-distance, round-trip send and receive, 4.8 years. That's about the time it takes for a bachelors degree student in the US to complete a bachelors degree. There's an enormous difference than between referring L1 in cache and doing a round trip overseas.

Latency and System Design



Okay, so let's try to pluck some of these numbers and look at them a little bit differently. We can put them, again, into these categories, memory, disk, and network. We can really start to appreciate the differences between these things. So here, you have durations in the order of seconds to minutes. Worse case, days. But in this case, you have the disk case, you have operations that take on your order of weeks or months And in the network case, you have operations that take a week to years. So, a round

trip on the same local area network is 5.8 days. So this is why the locality is important in distributed systems. You want to have two servers next to each other if they have to do some sort of network communication. You don't want one Server to be on the other side of the planet, because the difference here is between days and years. In any case, if you have to make some request to some service that you don't where it is, maybe it's on the other side of the planet. This could cost, relative to an L1 cache reference taking half a second, 4.8 years. These differences are enormous once they're humanized.

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.

Okay, now that we have seen this table of latency numbers that every program you should know, in particular with the humanized version of this table. We should now have some intuition about how expensive network communication disk operations really are relative to doing things in memory. So now that we have this intuition, now that we have this feeling of indeed how expensive these things are, one might ask, well okay great but how do these numbers about Latency relate to big data processing or big data analytics. So to try and understand how these latency numbers matter in big data processing let's have a look at the big system that everybody used before Spark appeared on the horizon. Let's have a look at Hadoop.

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.

So you may have heard of Hadoop before. Hadoop is a widely used large scale batch processing framework. It's an open source implementation of a system that Google introduced or that Google uses in house called MapReduce. MapReduce was a big deal in the early 2000s when Google began developing it and using it internally. Because it provided a really simple API. The simple map and reduce steps. So these kind of functionally oriented map steps. And, these functionally oriented reduce steps, where you kind of combined data. This was very easy for programmers to wrap their minds around, were large clusters of many machines. If all you had to do was think in terms of map and reduce and then suddenly you're controlling a lot of machines, but using these two operations, that's a really simple way to do these large scare distributed operations, but. Really, the big thing that it offered in addition to this API that lots of people could wrap their mind around easily, was fault tolerance. And fault tolerance is really what made it possible for Hadoop/MapReduce to scale to such large configurations of nodes. So that meant that you could do processing on data whose size was completely unbounded. Things that could never in anybody's wildest dreams be done in a week's worth of time on one computer, could be easily done on a network of hundreds or thousands of computers in a matter of minutes or hours. So, fault tolerance is what made that possible. Without fault tolerance it would be impossible to do jobs on hundreds or thousands of nodes.

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.

And the reason why fault tolerance is so important in distributed systems or in large scale data processing Is because these machines weren't particularly reliable or fancy, and the likelihood of at least one of those nodes failing or having some network issue or something was extremely high midway through a job. What Hadoop/MapReduce gave programmers was the ability to ensure that you could really actually do computations with unthinkably large data sets and you know that they'll succeed to completion somehow. Because even if a few nodes fail, the system can somehow find a way to recompute that data that was lost. So there two things, this simple API and this ability to recover from failure so you can then scale your job through the large clusters, this made it possible for a normal Google software engineer to craft complex pipelines of MapReduce stages on really, really big data sets. So basically everybody in the company was able to do these really large scale computations and to find all kinds of cool insights in these really large data sets.

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

Okay, so that all sounds really great. Why don't we just use Hadoop? Why bother with Spark then? Well, if we jump back to the latency numbers that we just saw, fault-tolerance in Hadoop/MapReduce comes at a cost. Between each map and reduce step, in order to recover from potential failures, Hadoop will shuffle its data over the network and write intermediate data to disk. So it's doing a lot of network and disk operations. Remember, we just saw that reading and writing to disk is actually 100 times slower, not 1,000. That was a typo. It's actually 100 times slower than in-memory. And network communication was up to 1,000,000 times slower than doing computations in-memory if you can. So that's a big difference. So remember these latency numbers that we just saw. If we know that Hadoop/MapReduce is doing a bunch of operations on disk and over the network, these things have to be expensive. Is there a better way? Is there some way to do more to this in-memory?

Why Spark?

Spark...

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

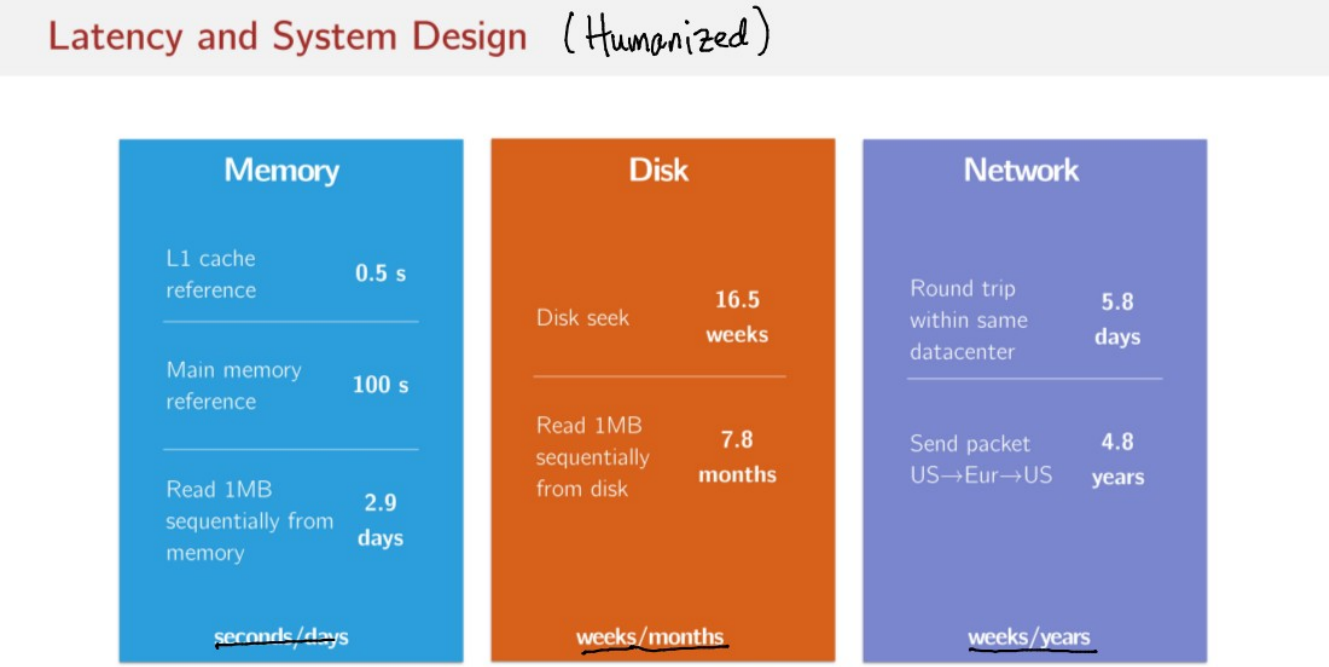
Achieves this using ideas from functional programming!

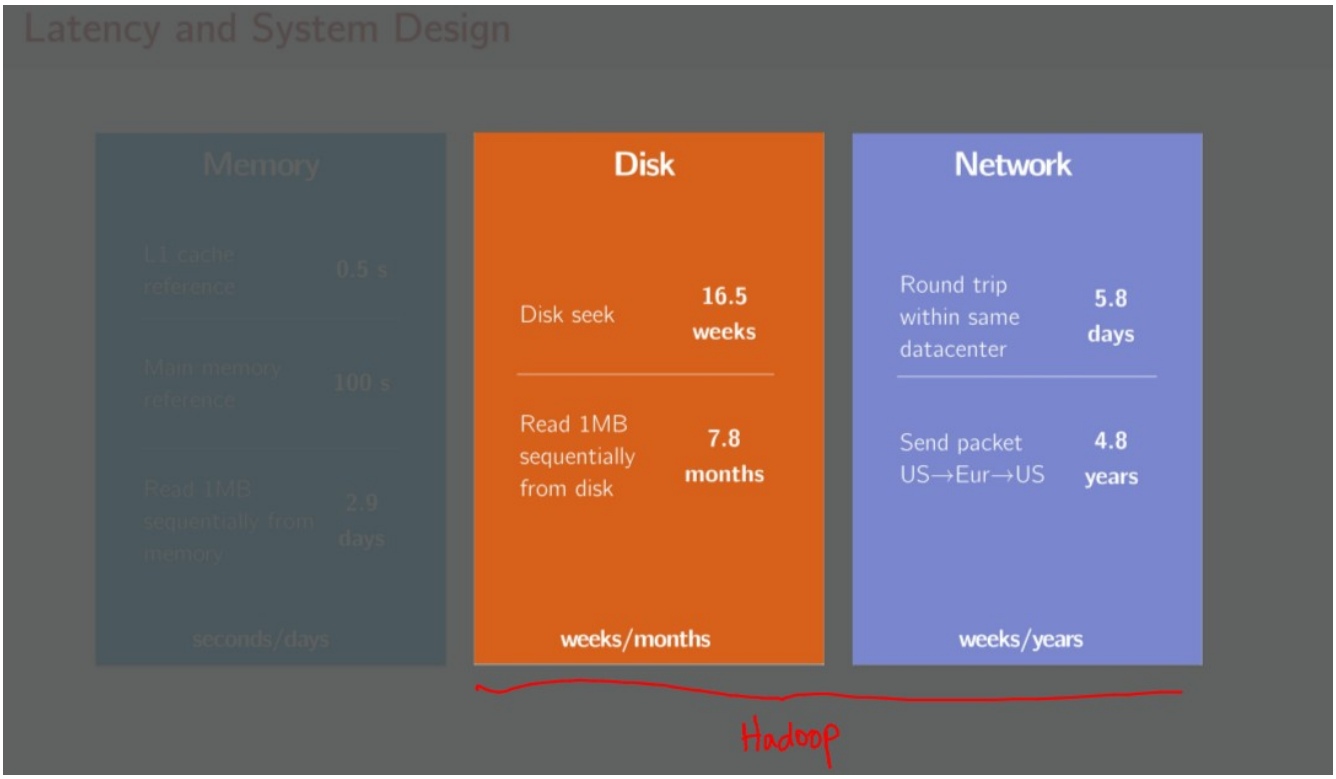
Idea: Keep all data **immutable and in-memory**. All operations on data are just functional transformations, like regular Scala collections. Fault tolerance is achieved by replaying functional transformations over original dataset.

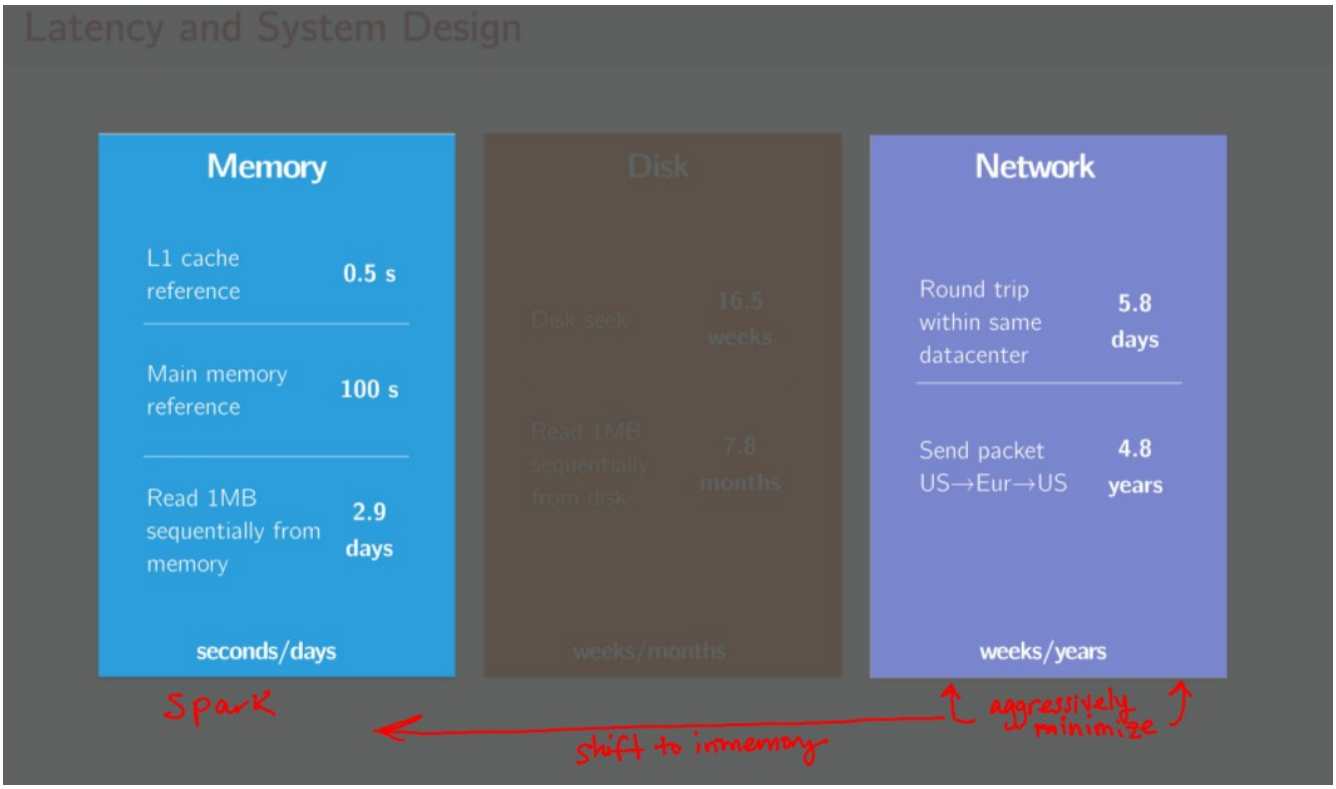
Result: Spark has been shown to be 100x more performant than Hadoop, while adding even more expressive APIs.

此處講得好, 是對 Spark 的好處 講得最好的. 以後寫 Spark 代碼時, 也要盡量用 immutable 和 functional transformations.

Spark manages to keep fault tolerance, but it does so while taking a different strategy to try and reduce this latency. So in particular, what's really cool about Spark is that it uses ideas from functional programming to deal with this problem of latency to try and get rid of writing to disk a lot and doing a lot of network communication. What spark tries to do is it tries to keep all of its data, or all of the data that is needed for computational as much as possible in-memory. And it's always immutable, data is always immutable. If I can keep immutable data in-memory, and I use these nice, functional transformations, like we learned in Scala collections, like doing a map on a list and get another list back. If we do these sorts of things we can build up chains of transformations on this immutable, functional data. So we can get full tolerance by just keeping track of the functional transformations that we make on this functional data. And in order to figure out how to recompute a piece of data on a node that might have crashed, or to restart that work that was supposed to be done on that node somewhere else, all we have to do is remember the transformations that were done to this immutable data. Read that data in somewhere else or on the same node, and then replay those functional transformations over the original data set. So this is how Spark managed to figure out a way to achieve fault tolerance without having to regularly write intermediate results to disk. So everything stays in-memory, and only when there's a failure is a bunch of network communication or reading and writing operations actually done. As a result, Spark has been shown to be up to 100 times more performant than Hadoop for the same jobs. And at the same time, Spark has even gotten more expressive(不是 expensive) APIs. So you're not limited anymore to map and reduce operations. Now you have basically the entire API that was available to you on Scala collections. This is also very cool. So you have something that is very fast, still fault tolerant, and on the other hand, you have even more expressive APIs available to you than you did in Hadoop. This is great.



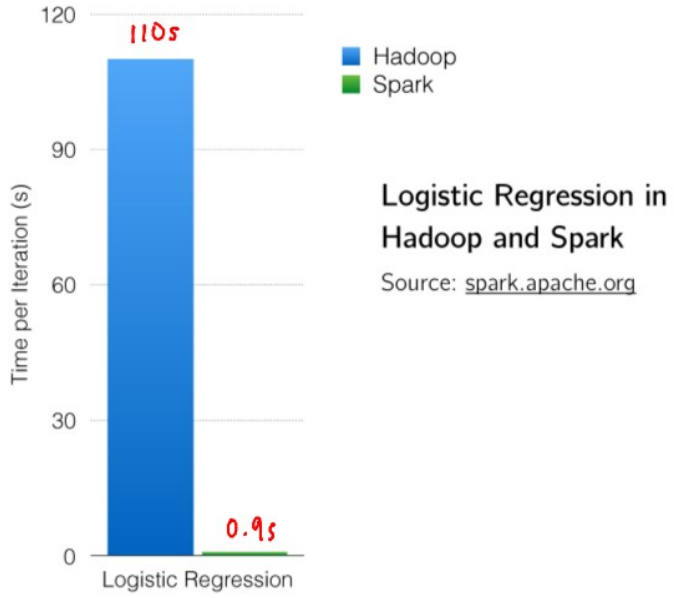




So to go back to this image that we drew, we've put things into categories, memory, disk, network, right? Network is weeks and years, disk is weeks and months, and memory is seconds and days. What

we have is this pattern, ultimately Hadoop sits here. So many of Hadoop's operations involve disk and network operations, right? So Hadoop is doing a lot of this stuff, whereas Spark is here. Spark is doing its best to try and shift many of its operations in this direction, more things being done in memory, less things being done on disk and over the network. As we'll see, it'll come out in the APIs that we're going to learn, but Spark does its best to aggressively minimize any network traffic that it might have to take. So we have this. We have Spark favoring in-memory computations and operations, aggressively minimizing network operations as much as it can. And Hadoop still doing a lot of its work involving the disk, and perhaps a little less aggressive, but [INAUDIBLE]. So just looking at these [INAUDIBLE] we [INAUDIBLE] Hadoop [INAUDIBLE] operates kind of on this side. Then we look at Spark here, we have more operations sitting on this side and fewer hopefully in the network box.

Spark versus Hadoop Performance?

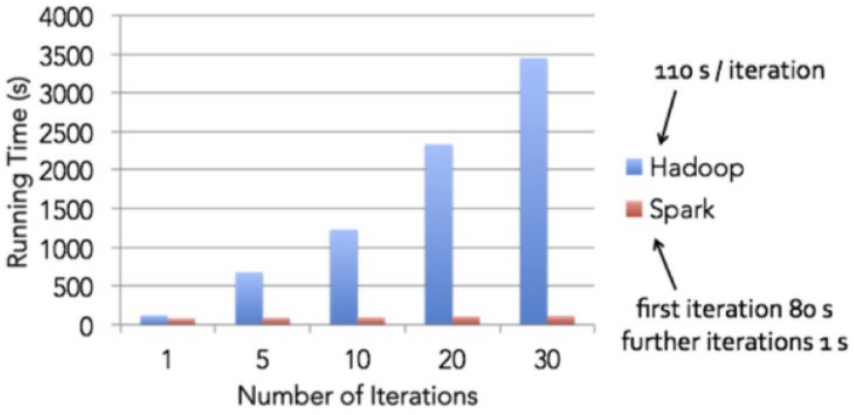


We have this trend. It's been shown on numerous occasions that Spark is significantly faster than Hadoop. So Hadoop, it takes 100 seconds per iteration, 110 seconds to be exact, per iteration for this Logistic Regression application, and Spark only takes 0.9 seconds. So in this case, it's about 100 times faster than this equivalent program in Hadoop. That's a big deal.

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



And just to show a few more numbers, related to this same application, logistic regression, what we see is that the first iteration that Spark does is kind of expensive. It takes 80 seconds for the first iteration but then the next iterations are 1 second. I hope this gives you some intuition about why Spark's different approach to handling latency is so important to, in the end, developer productivity. This enables developers to have a much tighter feedback loop where they can experiment with the same amount of data but only in few seconds to few minutes they can have a result, tune your algorithm and try again. Whereas if they were to use Hadoop for the same algorithm it would take them minutes to hours to see any sort of results and go back. Quantitatively, it's a lot of time the develops are saving by this clever approach to dealing with latency. And it's all built up and functional ideas, which is pretty neat.

Hadoop vs Spark Performance, More Intuitively

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

Hadoop/MapReduce

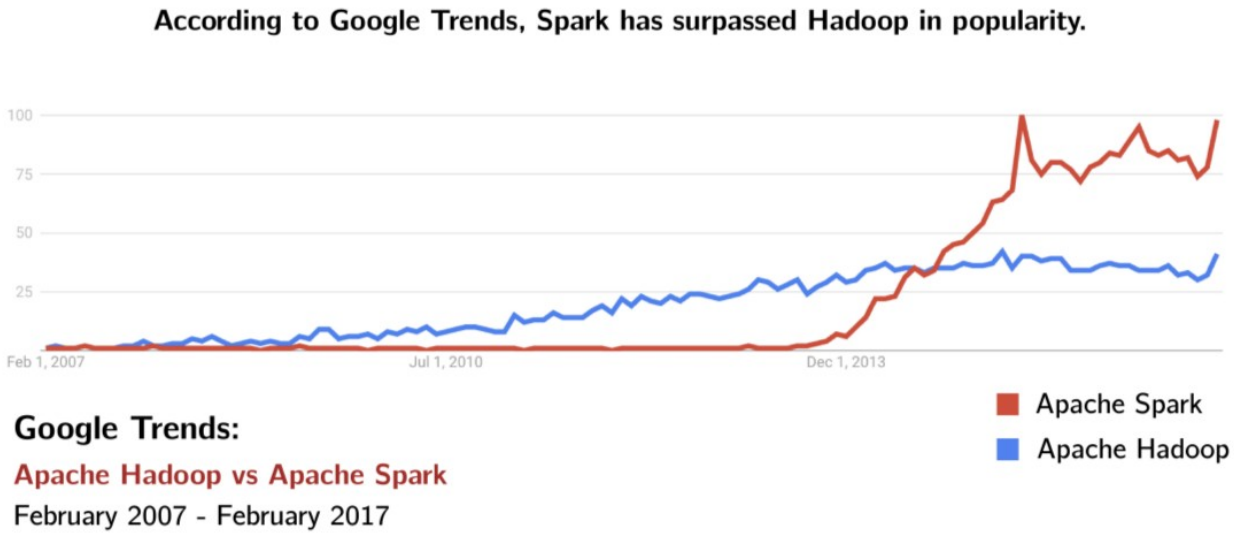
1. start job
2. eat lunch
3. get coffee
4. pick up kids
5. job completes

Spark

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

So intuitively, day-to-day these differences in performance, it has real impacts on on developer productivity. If you try to complete an equivalent job both in Hadoop and Spark, the same job in Hadoop you could start the job, go get lunch, then get some coffee with your colleagues, pick up your kids from school. And when you get home, SSH back into your work computer and see that the job just completed. Whereas Spark, you can start your job, go grab some coffee while your Spark job is running. And then by the time you're back at your computer, the job is finished. This means people are more productive. And you can get many more iterations done in one working day than you can solution built on top of Hadoop/MapReduce. So Spark delivers real productivity improvements to the analyst, to the developer.

Spark versus Hadoop Popularity?



And these benefits haven't gone unnoticed, you can check Google's trends. You can compare search queries for Apache Spark versus Apache Hadoop, and as you can see, since about 2013, early 2014, Spark has really experienced a massive uptick in interest, so much so that it's eclipsed search queries for Apache Hadoop. These benefits to developer productivity are driving the popularity of Spark, especially relative to Hadoop. People really see a benefit in this framework.

RDDs, Spark's Distributed Collection

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

Big Data Analysis with Scala and Spark

Heather Miller

In this session we're going to focus on Resilient Distributed Datasets or RDDs for short. RDDs are Spark's Distributed Collections obstructions and they're really at the core of spark. In this session we're going to learn the basics of RDDs and we're going to learn a little bit about the APIs that are available and we're going to consider some of the basic differences between the RDD, API and APIs that you may be more familiar with in 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 = ...
}
```

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.

So, off the bat, RDDs seem a lot like immutable sequential or parallel collections. So, think about for example a List. List in Scala are immutable collections and we often used them with high-order

functions like map, flatMap, or filter. Just like Lists, RDDs have a similar API, full of lots of high order functions like map, flatMap, filter, reduce. This is a simplified definition of the RDD class. Of course there are many more methods and it's much more complex than this. But this is a simplification of the RDD class as defined in the Spark codebase. And as you can see, these signatures all look familiar, they look just like the signatures that we've seen on List, for things like filter, flatMap and map. And it's a important to note that RDDs really make heavy use of higher-order functions. So, just as a reminder, higher-oerder functions are operations like map, flatMap, filter. These are methods or functions that take, as an argument, another function.

Resilient Distributed Datasets (RDDs)

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

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

So, just to look at some more methods on both RDDs and Lists, also other parallel and sequential collections. Map, flatMap, filter, reduce, fold, aggregate, all of these things exist as well on RDDs. So, we have really some of the core APIs still available to us, that we're already used to in sequential collections on RDDs.

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
```

They even have mostly the same signatures, macroscopically of course. So in the case of RDD is of course the return type is going to be in our RDD rather than a List, if we compare the signatures of map, flatMap and filter. Otherwise, everything is basically the same. In the case of map, both RDDs an List take up a function from A to B and apply this to all the elements in the collection. Same is true for faltMap and filter.

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
```

When we look at these reduction operations, same thing. We have effectively the same signatures. The only difference is aggregate. As we can see here, in the case of Scala's sequential imperial collection, aggregate has it's zero it's a buy in parameter whereas in Spark it's not. Do you have any idea why this might be? I'll give you a hint. But I'd like you think about it on your own and perhaps discuss it in the forums. Remember distribution, remember that we're going to send things to another computer. So think a little bit about whether or not a biname parameter is the best idea versus something that is not biname when it comes to having to distribute code.

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()
```

So I hope by just looking at some of these basic APIs, you'll agree that using RDDs in Spark it looks a lot like a normal Scala sequential or parallel collection. The only difference is that you know that your data is distributed amongst several machines. So assuming we don't know anything else about RDDs and we only know Scala's collections API, let's try to solve a simple problem with Scala's collection's API and see if it works for RDD's. So let's assume we have an RDD full of encyclopedia articles. Each encyclopedia article, each string represents the text of one article, for example, and let's just say, we want to search this RDD full of these articles or these article pages. And we'd like to count the number of times that EPFL, the university that the Scala Programming Language is developed at, let's say we want to count the number of times that the pages mention EPFL. What would we do? How would one write this code, knowing that we have something called encyclopedia, and we want to search for mentions of EPFL, and then sum up the mentions of EPFL? Well it looks very much like regular Scala collections. This is all you need to do. So assuming that each page is a string and all I gotta do is do page.contains my substring. So in this case, EPFL. All I gotta do is do encyclopedia.filter. I had this very simple predicate, and then I can just count the number of pages that I have EPFL in the text. That's it. Works exactly the same as how you would do it on a list.

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.

Let's look at another example. So word count is the hello world of programming with large scale data. How would we do word count in Spark? Hint it looks a lot like Scala collections. So let's assume we have an RDD. This is ann example of one more you can create an RDD. We will talk more on how to create an RDDs later. But let's say, I have created an RDD called RDD and now I want to count all the words in this RDD. The signatures are written here. This RDD has type, RDD (string) and let's assume that each string represents a line of text not text file. So what do we do? If we had a regular Scala list we could use flatMap to take each individual line and then split each individual line into individual words, now you have a collection of different words. Then we flatten that back down all to one collection. So the result of this here, as again an RDD of string. But this one represents now words. So it represents an RDD of all of the words. In this base RDD that contains lines of a text file. Next, what we can do is, we can map every single word in the RDD to a pair of word and the number 1. So we have something that we can now sum up and count. So then the next step would be to use a reduce operation. So, reduce by key is a little bit special for Spark. We'll look a little bit more into depth about exactly the differences later. However, what we're doing here now is we're just summing up the ones in the pairs, that's it. Now, we have a count of all the words in this text file. What's important to notice, that we've done this. We've done word count in only three lines of code. Another reason why people tend to favor Spark over Hadoop is that this program can be done in so few lines of code, so clearly, whereas, if one had to write this program in Hadoop it would take many more lines of code of lots of look like. So people find Spark so much more concise than Hadoop.

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`.

So last thing we're going to discuss in this lecture is there are different ways that one can create in RDD. So how might you create an RDD? There are two main ways that we can create RDDs. Either by transforming an existing RDD or from something called the SparkContext or in later versions of Spark it's being renamed to SparkSession. We have this thing called a SparkContext or a SparkSession, it's an object. And you can populate a brand new RDD with it. Either you populate a brand new RDD with this SparkContext or SparkSession, or you can get an RDD from an existing RDD. So, in the case of transforming an existing RDD, it's just like when you pull a high order function like map on a List. You get a new List back. In the same exact way, when you call a map, for example, on an RDD, you get back a new RDD. So, you can get an RDD as a result of doing a transformation on another RDD. Just like you can with Lists.

How to Create an RDD?

RDDs can be created in two ways:

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

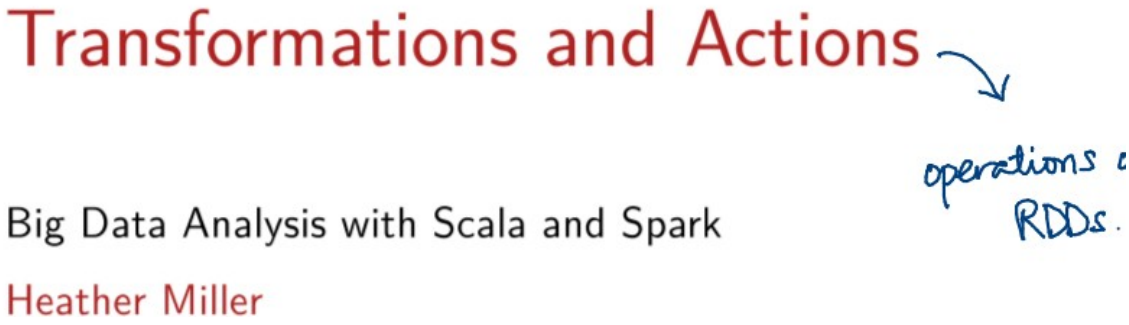

From a `SparkContext` (or `SparkSession`) object.
The `SparkContext` object (renamed `SparkSession`) can be thought of as your handle to the Spark cluster. It represents the connection between the Spark cluster and your running application. It defines a handful of methods which can be used to create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

Now the other way is to use this SparkContext or SparkSession object, as I mentioned earlier. This is important to remember because you're going to see this in your assignments later. Remember what this

SparkContext or this SparkSession object is because this thing, it might have a weird name, but this is the handle to this Spark cluster. This is how you talk to Spark. You talk to Spark always through this thing called the SparkContext. So whenever you start a new Spark job, you gotta create a SparkContext first. You gotta pass some parameters to it to configure your Spark job and then you've gotta start creating RDDs using this SparkContext. Now this SparkContext, a lot of the smart work surrounding distribution, figures out lots of things that you don't want to ever have to worry about. Most important to remember that this is your handle to your Spark cluster. And the two methods that are the most important to remember, that we're going to in all of the subsequent slides, and and in the assignments, are parallelize and textFile. Parallelize creates an RDD out of a regular Scala collection. You're not really going to come across this in real code bases where you're doing really big jobs because the assumption is that you already got the collection in memory, in a Scala collection like in a List or something else and now you just want to create an RDD. Most of the time, when you're using large data sets, you're going to be using this text file, a method, which basically can read a text file usually from HDFS, Hadoop's File System, or a local system like a regular text file and then return an RDD of Strings. So after you get this RDD of Strings back, then you need to figure out how to transform into something else. If you don't want it to be Strings, maybe you want it to be integers, or something like this. But the text file will return an RDD that you can then do all kinds of subsequent transformation and analyses with. In the next session we're going to delve a little deeper into RDD's and some of the most important operations that exist on RDD's, transformations and actions.

RDDs: Transformation and Actions



In this session, we're going to focus on transformations and actions.

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 are the different kinds of operations on RDDs. To understand transformations and actions and its work, first recall transformers and accessors from Scala's sequential and parallel collections. If you don't remember what these terms mean, I will briefly remind you. Transformers are operations that return new collections as a result. So, not a single value. If you, for example, call a method on a list and you get back another list, typically this is a transformer. So in this case, if we call a map on list, we get back another list. An accessor is a method that returns a single value as a result and not a collection. So examples of accessors are things like reduce, fold, or aggregate. So just to remind you what reduce does, it goes through and according to some kind of function you would give it, it combines all of the elements together and it will give you the single combined result back. This is an accessor.

Transformations and Actions

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

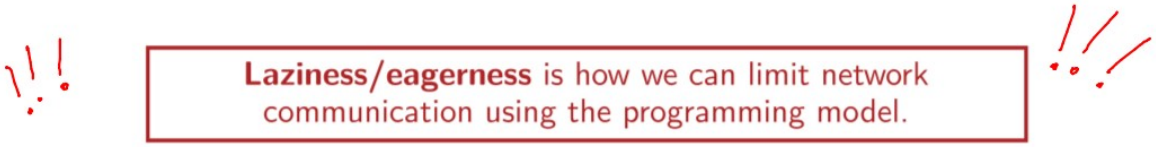
They seem similar to ~~transformers~~ and ~~accessors~~, but there are some important differences.

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.



Now, where we had transformers, transformers and accessors in regular Scala collections, we have in Spark transformations instead of transformers and actions instead of accessors. So the definition of a transformation, very similar to a transformer, is an operation that returns not a collection but an RDD as a result. And likewise, an action is something that computes a the result based on an RDD and it returns or saves that result to an external storage system, but it doesn't return an RDD. It returns something like a value, or something that's not like an RDD. But this is the most important thing, if you remember anything from this lecture, you need to remember that transformations are lazy and actions are eager. This is extremely important to remember. This is an enormous difference from Scala collections. So while you might have a method with a very similar signature like map, similar to the signature of map on Scala's lists, the fact that it's lazy and not eager is an enormous semantic difference that you need keep in mind when you are writing Spark programs. And the reason why this is so important is because this is the way that Spark deals with the network. The fact that transformation operations are lazy and actions are eager lets us aggressively reduce the amount of network communication that's required to undertake Spark jobs. So if you recall, in an earlier session, we were talking about how important latency was, and I mentioned at some point that this is going to sneak out into the programming model. This is how it gets into the programming model.

Example

Consider the following simple example:

sc → SparkContext

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

RDD[String]

RDD[Int]

What has happened on the cluster at this point?

Okay, so let's make that more concrete by looking at an example. So let's assume that we have a large list. And you remember, we're going to use this parallelize method. So I didn't define it here, but sc is a Spark context. We'll see later more concretely how to create an instance of these. But typically you always create a Spark context once in your program and then you keep reusing it, you create RDDs from it. And in this case, we can use that parallelize method we learned in the last session to transform this large list of strings into an RDD now of strings. Now this has type RDD of string. And so now that we have an RDD of String, we can do a map on it. We can go to each element and call length on each individual element, then we'll get back an RDD now of lengths. So now the type of lengthsRdd will be an RDD of integers.

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...

Okay, cool. So what's happening on the cluster now at this point? Well, the answer is nothing. Because remember, a map is a transformation method, it's deferred, nothing happens. All we get back is a reference to an RDD that doesn't yet exist.

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

So how de we ensure that this computation is done on the cluster? What do we do? Well, to kick off the computation and to wait for its result to be completed and returned, we can add an action. So in this case, we get the total number of characters in the entire RDD of strings. And we can do that using this reduce operation, which we saw on a earlier slide is an action. Reduce returns not another RDD, but a single value like an integer, for example. In this case, we're going to get back an integer. So this is important to remember. This is probably one of the most common problems that people have when they're just learning Spark. They erroneously assume that after a map or a filter operation has been done, that their computation has started or been completed. When in reality, nothing happens on the cluster. Nothing happens till invoking some type of action to return a result. So this is going to be very important for you remember in your first programming assignment.

Common Transformations in the Wild

LAZY!!

map	**map[B](f: A => B): RDD[B]** ← Apply function to each element in the RDD and retrun an RDD of the result.
flatMap	**flatMap[B](f: A => TraversableOnce[B]): RDD[B]** ← Apply a function to each element in the RDD and return an RDD of the contents of the iterators returned.
filter	**filter(pred: A => Boolean): RDD[A]** ← Apply predicate function to each element in the RDD and return an RDD of elements that have passed the predicate condition, pred.
distinct	**distinct(): RDD[B]** ← Return RDD with duplicates removed.

So just to give you a flavor of some of the more commonly used transformations that you'll see in Spark programs. So there are the usual suspects here, map, which, again, note that it's a transformation because of its return type. In of all these cases, an RDD is being returned. So these are transformations because the return type is an RDD. Which means these things are lazy, right? I'm just going to remind you again, lazy operations. So map we already know pretty well, apply a function to each element in the RDD and then return another RDD of the result of all of these applications per element. flatMap, same idea, but flatten the result. So return an RDD of the contents of the iterators returned. In the case of filter, we use this predicate function. And we return an RDD of elements that have passed this predicate condition in this case. It's a function from some type to a Boolean. And finally, distinct. This is also another rather common operation. And basically, this is kind of like distinct on a set operation. So it will remove duplicated elements and return a new RDD without duplicates.

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.

Now that we've seen a few common transformations that you're going stumble across in your assignments and in the wild, let's look at common actions that you might find in the wild. And again, I'm going to remind you with big letters that these are eager. These are how you kick off staged up computations. So some of the most commonly used operations are collect. So collect is very popular, because it's a way, after you've reduced your data structure down, let's say you did a bunch of filter operations and you've gone from some RDD that was 50 gigabytes, and now perhaps you've only got a handful of elements in it. This is how you get the result back. After you've reduced the size of your dataset, you can use collect to get back the subset or the smaller data set that you filtered down. So you get all elements out of the RDD. But typically you use collect after you've done a few transformations. You know your data set's going to be smaller, then you do collect to get all of those results collected on one machine. The next operator which we've used so far actually is count. So the idea behind count is very similar to the idea behind counts in collections API. Just return the number of elements in the RDD. Take is another very important action, okay? It's an action because we go from an RDD to an array. So you maybe have a very large RDD of 50 gigabytes worth of elements. And you say okay, take 100. And you get back then an array of those 100 elements of type T. So you basically are converting what was once an RDD into an Array. That's the same with collect, I didn't mention that, the return type of collect is an Array of T, because you're taking things out of an RDD and you're putting it into an Array on one machine instead of spread out on many machines in an RDD. Finally, we have reduce and foreach. Reduce I think we know pretty well, combine all of the elements in RDD together using this operation function and then return the result. So the return type of this is A instead of an RDD, that's how we know it's an action. And finally, foreach, because foreach returns type unit. So this applies this function to each element in the RDD. But since it's not returning an RDD, it's an action. So, again, how do you determine whether something is an action or a transformation? You look at the return type. If it's not an RDD, it's an action, which means it's eager. Never forget this.

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()
```

Let's look at another example. So let's assume we have an RDD of type String. So it doesn't matter where it came from. We have this RDD of type String now, which contains gigabytes and gigabytes of collected logs over the previous year. And each element of this RDD represents one line of logging. Okay, so this is a very realistic scenario which you might want to use Spark for. Perhaps you have many devices or machines constantly logging to some persistent storage somewhere like S3, okay. And now you want to analyze maybe how many errors you have in the last month. So assuming that we have these dates in a typical year-month-day-hour-minute-second format, and errors, when there is an error, the errors are logged with a prefix that includes the word error in it. How would we determine the number of errors that were logged in the month of December, 2016? Easy, we can start by calling filter. And on each line in the log, we can check to see if the string 2016-12 exists. And if so, only pass this predicate if also this is satisfie. So if error exists also in that line of text. So if you have both 2016-12 and error in that line of text, then filter it down into a new RDD with just these elements. So of course, nothing happens at that point. This is just a staged-up computation. And again, by staged, I mean it's a computation that we know we're going to eventually do, but we haven't started it yet. You could imagine that we could stage up many more computations as well. We could do another filter on this. We could do a map on it. We could convert everything in the logs to uppercase. We can do many more transformations that don't actually get executed yet. We can just stage them up, we can queue them up, okay. And finally, in this case, we'll call count. That actually gives the order to Spark to send this function over the network to all of the little individual machines to do their computations, and then to add them up and send back the results, the count call. And to aggregate it, combine it all up, so that you have one integer or one long with the number of errors in the logs. I bring up this example because it illustrates why laziness is useful. So even though this is very simple, it stills illustrates something that is really useful about doing this in a way where transformations are lazy and actions are eager.

Benefits of Laziness for Large-Scale Data

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

This helps when processing large amounts of data.

filter
↓
take

Example:

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

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

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

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

By staging up computations in this way, we can optimize them. So Spark can be very smart and decide when it can stop doing work to save time. So in a very similar example, it's basically the same example, where we have an RDD of strings, we do a filter, it contains an error, and then we do an action, in this case take 10. You could imagine that it's exactly the same code as here. What's really happening is that the execution of filter is being deferred until this take function is applied. And this means, Spark can be very smart. Spark can stop filtering elements once we've gotten ten of these. So Spark does not have to first go through everything, make a new RDD all over the place, all over the whole network with all of the instances of error, it can just stop when we've gotten to 10. So this is beneficial because it saves time and work, and it can do smart things like not computing intermediate RDDs. This is why it's advantageous that transformations are lazy, because then we can do all kinds of optimizations on them.

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.

Finally, I just want to run through a few other kinds of transformations that you might encounter in this course. There are a special group of transformations that combine two RDDs. These are typically set-like operations like union or intersection. So if I have an rdd1 and an rdd2, I can create the union of those two RDDs by saying val rdd3 = rdd1.union(rdd2). In this case, rdd3 will be a new RDD containing the elements from both rdd1 and rdd2. So these are regular set-like operations. And the same goes for intersection. So you can do the same thing with two different RDDs, or you can return a new RDD containing the elements only found in both RDDs. Subtract is also quite useful, it's the same idea. But you can subtract the elements out of another RDD. This is kind of like when you're using a vector editing program and you can also subtract shapes from one another, right. And then finally, there's the Cartesian product with other RDDs. And again, you know that these things are transformations because the return type is always an RDD. So it's lazy.

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.

Finally, there are a handful of actions that you're going to come across. These are actions that don't exist in Scala collections, surely there are some things that are different between Scala collections and Spark. These are operations which are useful when dealing with distributed data or very large data sets. So just to repeat, we know that these are actions because the return type is not an RDD, which means that they're eager. Okay, so operations like takeSample. If you want to sample down or decrease the size of your data set, takeSample will give you a random sample of a certain number of elements that you want, but it returns an array to you. So you take a very large data set, you sample different pieces of it, and then collect the subsample data set to one array on one machine rather than have it be distributed in RDDs. Likewise, there's another operation called takeOrdered. So you can, in this case, you can do a take, but have them be ordered. So you can return the first n elements of an RDD using either their natural order or a custom comparator. So that's this implicit ordering parameter here. And finally, there are two different kinds of save as functions. So this is of course very important when you're doing distributed jobs. So these are side effecting operations. You can call saveAsTextFile on an RDD. It's an action. You had queued up many transformations, and then now you say, okay, save as a text file. It causes all these of transformations to be computed, and then this thing can be written to file at this path. This lets your write to a file either in the local filesystem or in HDFS. And saveAsSequenceFile is very similar. So you can save the elements of your dataset as a Hadoop SequenceFile in the local filesystem or HDFS. So again, these are actions. And these are things that are very practically useful and that you might come across in this course or in Spark codebases elsewhere.

Evaluation in Spark: Unlike Scala Collections!

Evaluation in Spark: Unlike Scala Collections!

Big Data Analysis with Scala and Spark

Heather Miller

In this session we're going to talk about evaluation in Spark and in particular, reasons why Spark is very unlike Scala Collections. So as I've already pointed out in previous sessions, there's this laziness eagerness thing going on between transformations and actions. We covered some simple examples which showed how evaluation was different in Spark. These differences can compound and link to inefficient programs. So it's very important to understand completely how your Spark programs are being evaluated because there are many situations where you want to avoid unnecessary evaluation in Spark. So in this session, we're going to look at exactly how evaluation works and some common scenarios you should think about when writing Spark programs.

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!)

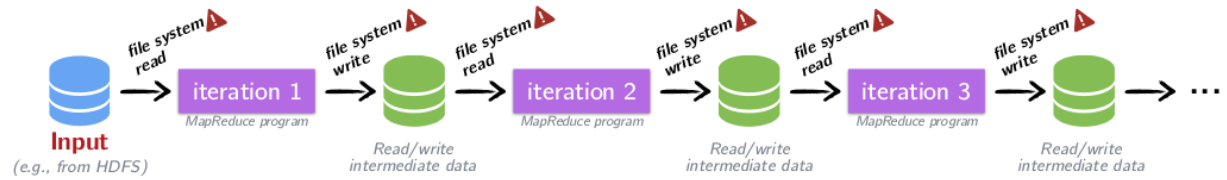
Why do you think Spark is good for data science?

Hint: Most data science problems involve iteration.

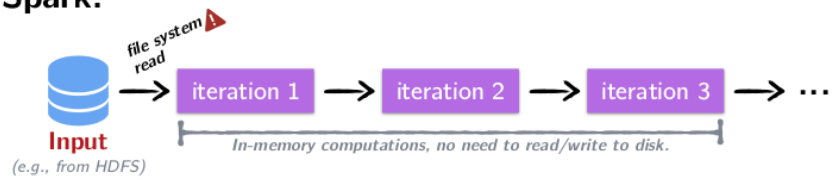
So let's first start with a little bit of a digression(離題). So why is Spark good for Data Science? In order to answer that question, let's start by recapping some major themes from previous sessions. So in the last session we learned about the difference between transformations and actions, in particular that transformations are deferred or lazy and actions are eager. They kick off stage transformations. So we learned that last session and a few sessions earlier, we learned that latency is important. It makes a big difference in the experience of the actual data analyst using Spark and that if we can shift things in memory as much as possible, we can significantly lower these latencies and save a lot of time and improve the experience of the analyst analyzing data. This analyst can get more work done, basically, with lower latencies. There are these two points that we've learned in previous sessions. And somehow they're going to come together into an answer. But why Spark is good for Data Science? Can you see why? Is it clear yet? I'll give you a quick hint. Most data science problems, when you think about them, they involve some kind of iteration (iteration 就是 Java 裡的 iter.next() 這種東西). So how do transformations, actions, and in-memory computation, how are these things somehow important to data science problems that involve iteration?

Iteration and Big Data Processing

Iteration in Hadoop:



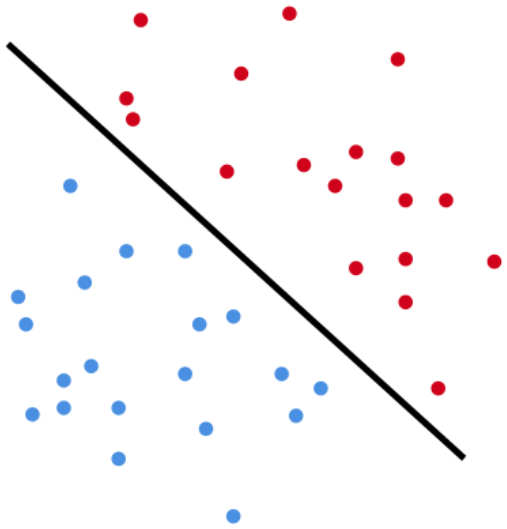
Iteration in Spark:



Well first, let's look at Iteration in Hadoop. So here's a graphical illustration trying to show what an interative program would look like. A program with at least three iterations, probably more. In this example what we have is some kind of input, perhaps some data in HDFS. And then we have iterations that are implemented as little map produced programs and we iterate by running these individual map produced programs many many times. So at each iteration we probably have a map at a reduced step. And between iterations, we want to write the data that we computed, and then we want to read it again for the next iteration. So we know that there's probably reading and writing going on in each of these iterations. And there's also reading and writing going on between these iterations. So we have to read and write this intermediate data, each time we want to end or start a new iteration. So one thing that might jump out at you if you recall the latency lecture, is that a lot of time is spent then in IO, disc IO or even network IO. One figure has attributed around 90% of this time being spent in IO operations, which Spark can avoid. So if you want to do iteration in Spark it can be much more performant because all of this reading and writing to disk can be completely avoided in Spark. We have to of course read in data from HDFS. But once things are in memory we can just iterate on it in memory. We can run the same sort of map reduce steps, or filter or whatever steps, many times, again, and again, and again, for how many iterations that we want. And it's all sitting in memory. It can still be recovered if there is a failure. And there's a lot of reading, and writing, and things that just don't have to happen because we don't need to persist things to disk all the time. So begin to answer this question about why Spark is good for data science applications and why transformations, and actions, and in-memory computations are important? We can start to see here that it's better for iterative programs to be done in memory because they'll be faster. But what does it have to do with transformations and actions?

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.



To understand why transformations, or actions, have to be carefully thought about in this situation, let's look at a quick example. So let's look at logistic regression, which is kind of the Hello, World of big data iterative algorithms. Logistic regression is a popular iterative algorithm that's typically used for classification tasks. So assuming you have two different Classes of data blue and red. And what we want to do is we want to come up with a classifier or a way to separate out these two different classes of data. So the classifier will be basically this line. We move it around on this space here. Until it separates this two classes of data. And then if we ever get new data we assume that it follows the same trend as the red and blue and hopefully it should work for any new data that's added right. That's the idea of a classifier. So just visually, to get an idea of what we're doing here. On one iteration, we moved the line a little bit. On another iteration, we moved the line a little bit. We keep doing this, and like I said, we're iterating through all the data every time. We're doing some math with all the data. And then eventually we find a good separating line here. And the algorithm stops. And what we're focused on is less the details of logistic regression. But we're really focused on the shape of the computation, the fact that we pass through all the data on many iterations to come up with a new estimation of this line separating two sets of data. That's the shape of the computation that we're going to be looking at.

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?

And this is how we're going to update those weights that represent the position of that line that separates the two classes of data, right. So, as you can see here, there is a number of iterations that we're going to do and we're going to iterate through these x, y data set. And there's some function g which doesn't need to be shown here but it's part of the logistic regression algorithm. So this algorithm can be implemented in a very straightforward way in the Spark. Here it is. So just to step through it, what we have here is first we have an RDD points and this is read in from textFile somewhere and then this textFile is parsed so we have a map function that goes through all blinds of the textFile. Remember textFile reduce RDD full of strings so we have to convert all these strings into basically doubles here. We have a vector of weight, so these Ws here which we initialize as a 0. Then for some number of iterations that we determine elsewhere, we go through the entire data set several times. So here we go through all of the points and then we apply this G function here, okay? And then we reduce, and now we update the vector of weights here using what we have just computed, using this gradient that we've just computed, okay? Okay, cool. So now we understand the basic shape of this algorithm. It's pretty concise isn't it? Essentially we iterate on applying a map and the reduce for some number of times on a certain RDD, we apply math and then redo. So we do this perhaps 30 times. And you update some local vector with the results every single time you go through this entire dataset. Okay, so that's in general shape of this algorithm. But what's really happening here? Remember, on the first slide, we were talking about trying to keep computations in memory. And having to think about what are transformations and what are actions? So what's, what's really going on here? I'll give you a second to think about it, and then we'll talk about it on the next slide.

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?

Well, the short answer is that we're doing a lot of work that we don't have to do. Remember, transformations like map get reevaluated every time an action is used. So we do have one map here which gets re-evaluated when we call this reduce here. We also have a map here. That means we're applying this map function again and again and again when we don't have to. So points is being re-evaluated on every iteration. If we have 30 iterations for re-evaluating points, 30 times, that's completely unnecessary, we shouldn't have to do that, that's a lot of extra work that we don't want to be be doing right now. It would be much better if what we could do is create these points RDD, leave it sitting in memory because we that's faster and then reuse it on every iteration. Instead what we're doing is we're recomputing on every iteration which is surely very expensive.

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.

cache()和 persist()功能稍有區別, 後面會講

So we see here a major consequence of these lazy symantics of these transformation operations. By default, RDDs are recomputed each time you run an action on them. That's something you can't forget about, and of course as we just saw especially in the case of an iterative algorithm, this can be very expensive in time if you need to use a data set again and again. But what's nice about Spark is that it gives you a way to control what data you have cached sitting in memory, available to reuse again. In order to instruct Spark, hey, I'm actually going to use this again, don't recompute it a thousand times, you can use a method, persist or method cache. What these methods do is after an RDD is evaluated, it keeps them in memory, so you can reuse them again.

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.

So here's a quick example of a situation where you might want to use this cache function, or this persist function. So recall this example where we have an RDD full of strings that represent years worth of logs. And let's just say we want to go through all of these logs and we want to filter out the ones that are actually errors, right? And we know we're going to have to use this a few times, so here we have an action take. And in this case so far, if we didn't use persist, this would be okay. What we would be doing is we would be evaluating it once at the point which we call take(10). And that would be it. We wouldn't need to evaluate filter a bunch of times. Because we only really use it once.

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.

However, if I add a line of code here. Here's another action. So I have now take and count, it's beneficial now to persist this LogsWithError RDD because if we don't persist this, if this is not cached in memory, then what happens is I do this filter on lastYearsLogs twice. I do it when I compute take(10) and I do it when I compute count. So, what we can do is if we do persist, that means that this logsWithErrors, this RDD sits in memory, and it can be reused. So it can be, we just use that again without having to recompute this filter for the take(10). And then we again reuse the same RDD that's sitting in memory to compute this count without having to go through that filter again.

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.

So if we go back to this logistic regression example that we just saw, we can add persist to this points method here and what that will do is that will prevent us from having to evaluate it on every iteration. We evaluate now, we just evaluate it once and then we re-use that data set that's sitting in memory each time. We do this map and this reduce on this points data set to compute this gradient on each iteration. So here, we can save a lot of effort and not have to continually re-compute this RDD by caching it in memory the first time we use it.

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`.

Okay, to dive just a little bit deeper into caching and persistence in Spark. It's possible to persist your dataset in many different ways. For example, as just regular Java objects sitting in memory so, just Azure program normally run, nothing special just to keep your data in memory. You can also persist to disk as regular Java objects. You can persist in memory as serialized Java objects, so that is converting these objects and memory to, byte arrays or arrays of bytes which is much more compact but it takes some effort serializing and deserializing. So it saves memory but it costs a little bit of compute time. Same thing put these on disk or a mixture of keeping data in memory and on disk. And in particular, in this case, rather than re-evaluating many transformations a bunch of times, anything that needs to spill over to disk, in case you don't have enough memory. Everything that you can keep in memory, you keep in memory, but if you don't have enough memory, then we persist the thing that you tried to cache to disk. That's actually very different from this here. If you run out of memory in this case (第一個, 即 in memory as...) , you don't have enough memory between all of your nodes to hold the data set in memory. What happens is a least recently used policy for evacuating things out of memory. So let's say you have an RDD. Let's say you've done 20 transformations on this and it takes a long time to go through all of them, right? If you don't have enough memory to hold that resulting data set, you can persist on it and you've chosen this mode (第一個, 即 in memory as...) here. What happens is Spark only holds as much as it can in memory and anything else that it can't hold in memory, it throws it out. If ever those pieces of data need to be reused, then it will re-evaluate them, it will go through all those many transformations that are required to reevaluate the missing pieces of data it doesn't have. So in this case, this right here could be very useful in a situation where it takes a lot of work to get to whatever thing you try to persist and then can persist at least the thing that you've already computed. So just to give some context, there is many different ways you can configure, actually help the systems

works in your smart job. And the two methods that you use are cache or persist. Cache is a shorthand for default storage level. So storage level is what we use to configure these things but you can just write cache with some parentheses behind it. And that will be the default storage level which is memory only for regular job objects. That's this one here. And persist is another method that you can pass different arguments too representing one of these so called storage levels in order to customize how the persistence works. So this is the method you use if you want to change the default behavior.

Caching and Persistence

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

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

Default

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

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

And just to show you a quick chart of the different storage levels that are available, these are the things you pass through the persist method and here's just the chart kind of showing the pros and cons of different storage levels. So for example, MEMORY_ONLY uses a lot of space, but it's pretty fast because you don't have to do any extra processing on this data. It's always in memory, never on disk. Whereas some of these other ones like MEMORY_AND_DISK and MEMORY_AND_DISK_SERialized and what not, again they have different trade offs. So, when you keep things serialized, it costs a bunch of time serialize and de-serialize it CPU time. But of course it uses very little space in memory, so you can put more data in memory but of course you're going to have to serialize and deserialize each individual element of that data set if you ever want to access it. So, there are different trade offs, this is for you to decide depending on the job that you're doing. But just be aware that there are a lot of choices, but in general, throughout this course we're always really just going to use the default. So we're going to use this memory only storage level, which you can get by just calling cache or persist without arguments.

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.

So key takeaway of this session is to remember that despite how similar Spark looks to Scala Collections, this is deferred semantics of the RDDs that you're using, have a very different behavior compared to Scala Collections. So the need to use this cache function to prevent things from being evaluated multiple times. For example, this is completely unlike Scala Collections even though the same interface is available to you. And you got to remember this because this could be the difference between you waiting a really long time for a job to complete or a job being finished in a few seconds. So just to reiterate due to the lazy semantics of these RDD transformation operators and due to many users sort of implicit reflex to assume the eager semantics of the regular Scalar collections. One of the most common issues that newcomers have when they start using Spark is this issue where people are unknowingly re-evaluating several transformations. So you're just kind of doing this in computations again and again and again without realizing that you're doing these mini computations again and again and again. And you could save all that effort by doing caching. And of course this is something important to point out when you do your programming assignments, this is something that students mistakenly do all the time. And as a result during the programming assignments lots of students have a tendency to run out of memory and to not know why, this is a big hint. This is probably going to be the main reason why you might be running out of memory 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)
```

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`.

So just before we conclude, I want to remind you that actually this laziness is really useful. Because after the last slide, you must be like my gosh, I have no idea what's happening under the hood. How do I know if things are being reevaluated all the time? I have to look much more closely than I thought at my Spark programs. Which might seem like a bad thing off the top of your head, but [COUGH] I will remind you again, via two examples real quick, why laziness is a good thing. So we saw this example actually in the last session, just to remind you. Let's assume we have an RDD full of strings which represent a year's worth of logs. And then we do a filter on it to filter out the logs that have errors and then we just take the first ten out of those, right? So why is laziness useful here? It's because filter is delayed until take is applied, that gives Spark the opportunity to inspect this chain of operations that are sort of queued up before being executed. Let Spark figure out how to do important optimizations. So for example on this case, what Spark can do is it can say, so I only need to find ten results that match this predicate here from the filter because I'm only going to take ten of those. So rather than having to go through the entire data set, Spark can intelligently just say, okay once I've got ten items that match this predicate here, I'm just going to quit and save a whole bunch of time. And not have to iterate through the whole data set just not to use it right? So Spark can do important optimizations like these.

Restating the Benefits of Laziness for Large-Scale Data

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

Example #2:

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

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.

注意本情況無 take(), 故要 traverse 整個 RDD.

So another important optimization that Spark can do is fusing transformations together. So just like before, Spark now has the opportunity to make an optimization before the count is actually executed, and in this case what Spark can do is it can stage these computations and then fuse them together so it has to do less work. So in this case, because map usually requires that you do one entire traversal through a data set and filter also requires you to do one entire traversal through a data set. Spark can say, hey look, I've got two traversals that I've gotta do. Maybe I should only traverse this data set once, rather than many times. And I can compute the result of both the map and the filter in a single pass before returning the count. What Spark can do here is it can basically optimize this, and then instead of pass through the data set many times, like would be the case for eager collections, like Scalar's regular collections, and Scalar's regular collections would be passing through the data set many times. Spark can very intelligently say, I don't have to do that, I can do less work. So laziness, remember, it's a very good thing. You just gotta remember that you have to think about it when you are writing programs. You can't just write code and imagine that everything is eager. You have to remember that transformations are lazy.

Cluster Topology Matters!

Cluster Toplogy Matters!

Big Data Analysis with Scala and Spark

Heather Miller

In this session we're going to talk about the anatomy of a Spark job. We're going to look at how clusters are typically organized that Spark runs on. And this is actually important. It's going to come back to the programming model once again. You can't just pretend like you have sequential collections that are on one machine. You actually have to think about how your program might be spread out along the cluster.

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?

So, let's start with a simple example to try and motivate this. Let's say we have a case class person. It's got a name, which is a type of string and an age which is typed Int. And let's say we have RDD full of these person objects. So, we have an RDD called people. Now if we say people.foreach and we println, so just want to print out all of the people in our RDD. What happens? I'm going to give you a minute to think about it or maybe even to look back at your previous notes.

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)
```

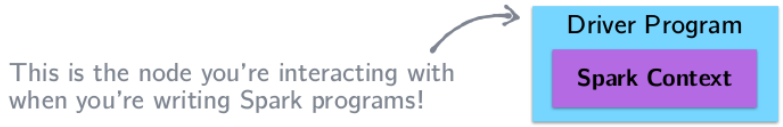
Where will the `Array[Person]` representing `first10` end up?

Actually before answering that question let's look at a second example. Assuming again we have an RDD full of these person objects, and I say people.take(10) what happens? Where will the array of person objects that is returned by take(10) end up? I'll let you think about this too.

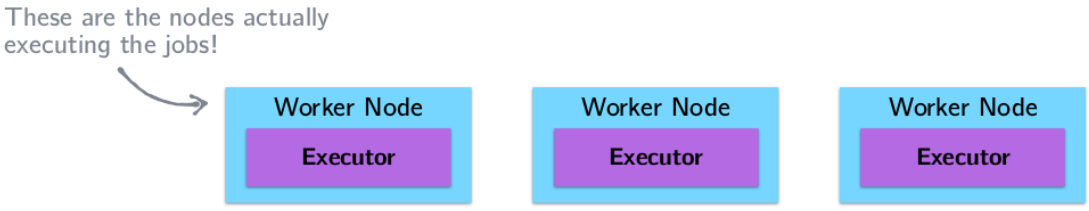
How Spark Jobs are Executed

Before I give you the answer for both examples let's first look at how Spark jobs are executed. Spark is organized in a master worker topology. Remember you are writing a program which is then getting distributed to a bunch of workers where your data is. And typically roughly the same operation is being done across all of the workers. So usually there's one master and many workers. In the context of a Spark program, we refer to the node that acts as the master as the driver program. And we refer to the workers as the worker nodes.

How Spark Jobs are Executed

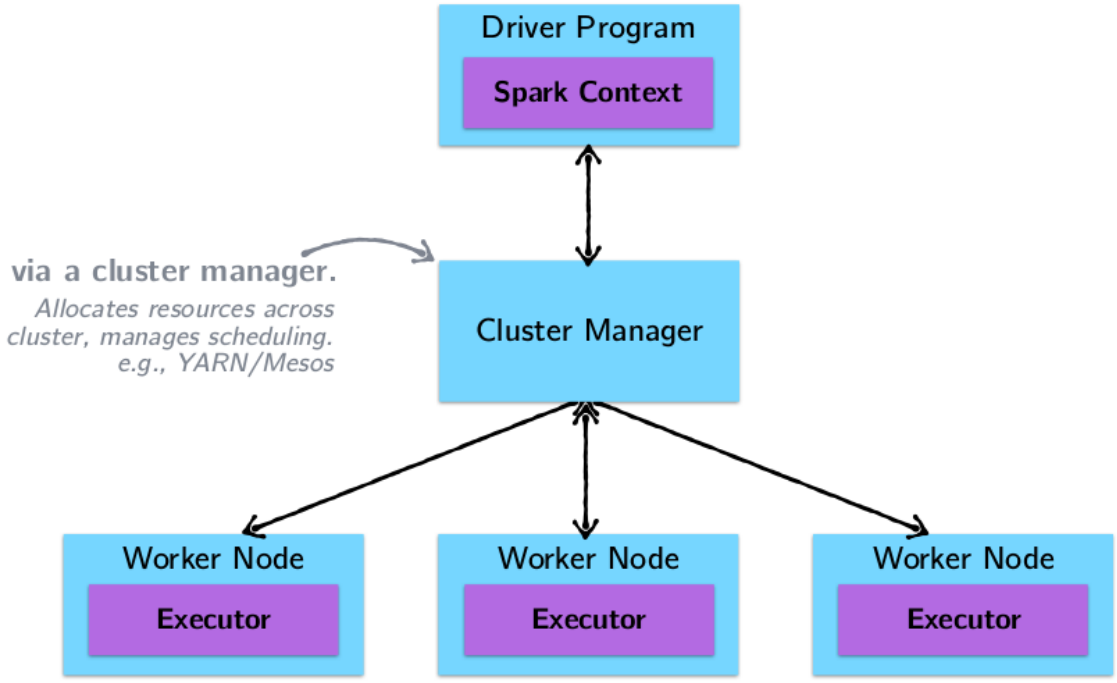


But how do they all communicate?



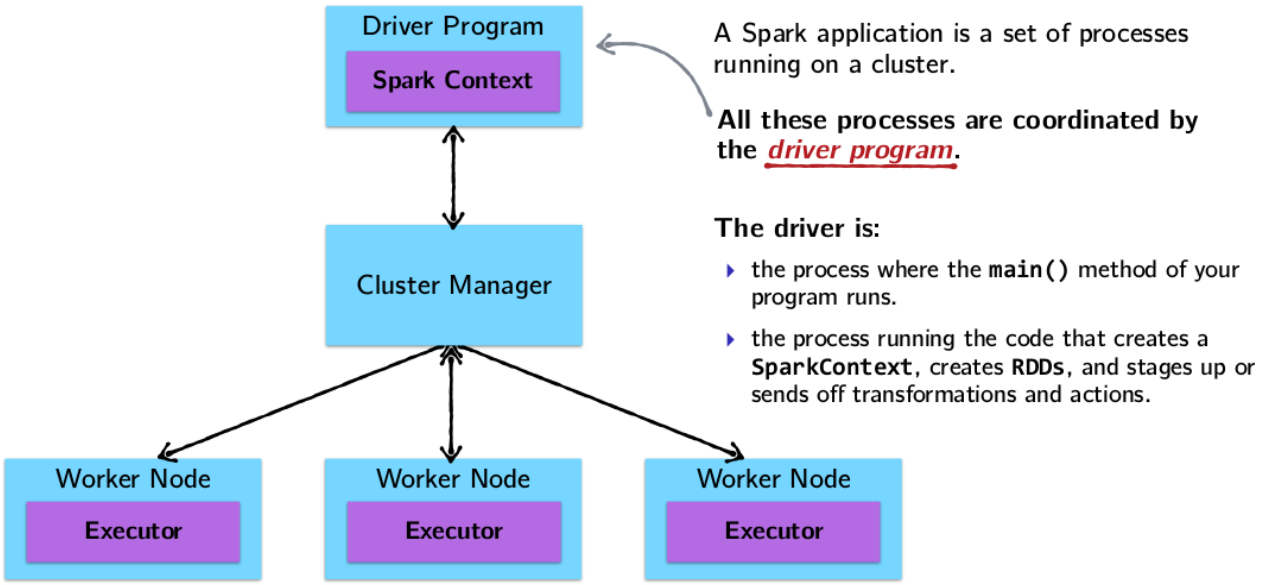
Now the driver program is where the spark context is. So this is the thing that we are using to create new RDDs, to populate new RDDs, and when we do Spark context or SC.paralyze or SC.textfile, we're actually using this thing here in the driver program. When you write a Spark program, you write a Spark program from the perspective of this node here. You think as if you were the driver node and you're giving commands to worker nodes. This is a node that you're interacting with when you're writing Spark programs, and these are the nodes that are actually doing the jobs. So the executors are actually doing all the work, and this is coordinating the work. You can think of it that way.

How Spark Jobs are Executed



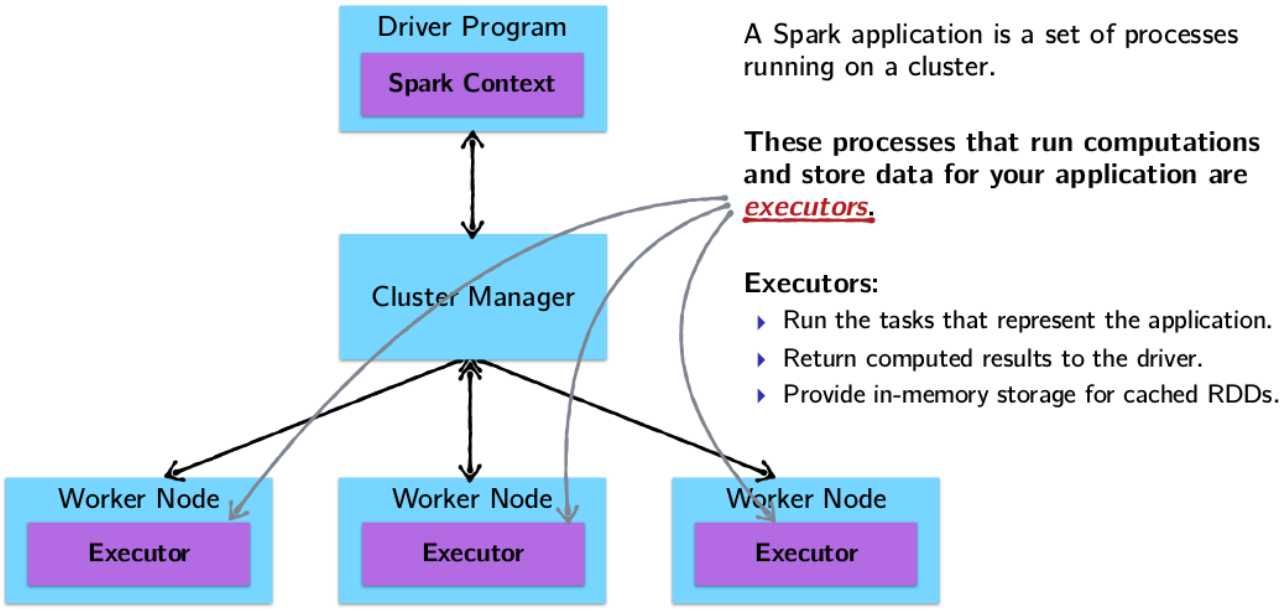
But of course, how do they communicate? They communicate via something in the middle called the cluster manager, and a cluster manager is what actually does the work of doing all of the scheduling, and managing resources across the cluster. It keeps track of everything. It does a lot of bookkeeping. Popular cluster managers are YARN or Mesos. So often when you're running on a real big cluster that you're maintaining yourself, you typically choose one of these cluster managers to run on top of.

How Spark Jobs are Executed



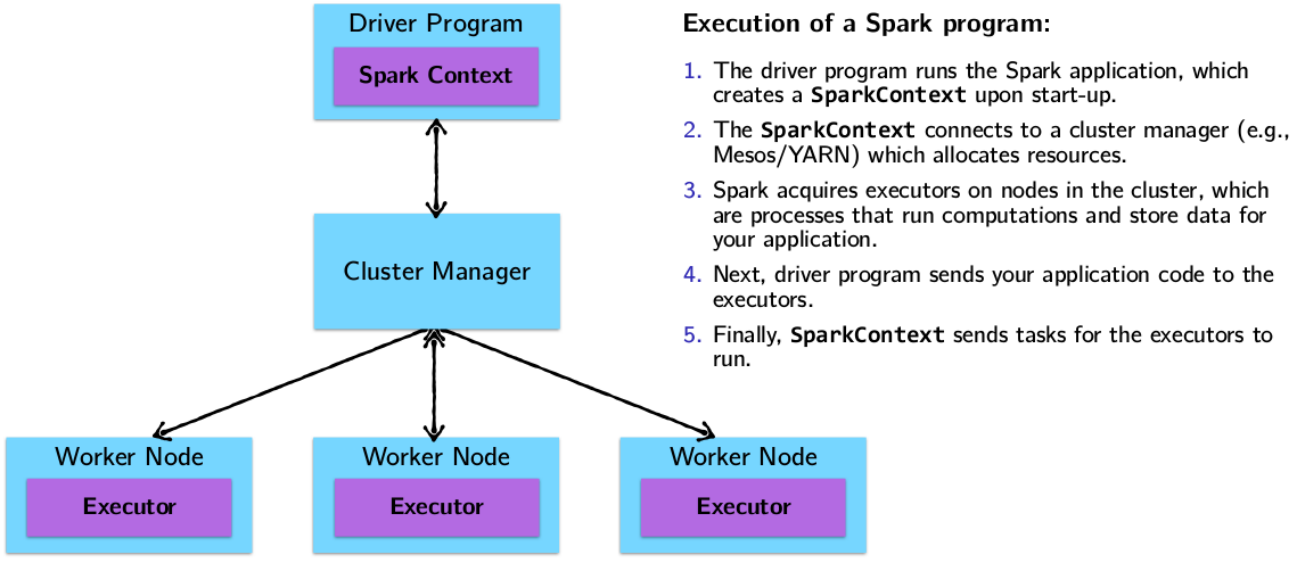
Okay, so this is more or less the anatomy of a Spark job. So you have this driver program, you have your Spark context in it all of it is kind of passing through this cluster manager thing that takes care of stuff like memory and worker node and all of the work is being done in these things called executors on the worker nodes. So when you take a step back, it's best to think of a Spark application as a set of processes that are running on a cluster where the Driver Program, the driver node here, the master node, is coordinating all of these processes. And it's important to remember that the driver is where the main method of your program runs. And the process running the code that creates the Spark context creates RDD's and it stages up and it sends off transformations and actions. That all lives here on this driver node, so the driver node or the driver program is really the brains of your Spark job.

How Spark Jobs are Executed



So these processes that make up a Spark application, these are all running in these executor things here. And this is also where your data is stored and cached. So just to summarize, executors run the tasks that represent the application, they return the computed results to the driver so whenever an action is done on one of these executors, the executor is the one that returns the result back to the driver program. And of course, executors also provide in memory storage for cached RDDs.

How Spark Jobs are Executed



So to summarize the steps that represent the execution of a Spark program, the driver program runs the Spark application, which creates a SparkContext upon start-up. Then the SparkContext connects to a

cluster manager, for example, like I said, the most Popular are Mesos/YARN which then allocates all the resources everywhere. After that Spark requires executors on nodes in the clusters. Now that you have a few nodes and you have this clusterizer running, then you establish executors everywhere. And these executors are the processes that run computation and store data for your application. Next, your driver program then sends your application code to the executor. So the driver program is actually sending code around. Think about it, you're sending a function around to the individual nodes here. So if I want to do a map on an RDD, this guy is actually sending that function to all the individual worker nodes and saying, hey apply this function to the data that you have on your node and then maybe there's an action, and then send the result back to me. And finally the SparkContext sends tasks for the executors to run. So the SparkContext figures out what to do and then sends it along to the executors to actually execute that work.

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.

So let's go back to our first example. Now that we know a little bit about the anatomy of a Spark Job. So remember we had an RDD of these person objects, we did people.foreach(println). So the question was, what happens? Well, on the driver nothing happens, why might that be? I'll give you a hint, foreach is not a transformation. Remember that foreach is an action and it has return type unit, so it's a side affecting thing. Therefore, because it's an action, it's eagerly executed on the executors and it's not being executed on the driver. Any calls to println that happen inside of this foreach thing, end up happening on the individual worker nodes. So then, whatever this println is doing is being executed on the worker nodes and it's going to the standard out of the working nodes, which you can't see if your looking at the standard out of the master node that you're on. So, basically, what this does, is it prints lines on the workers that you can't see because you're sitting on the master.

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.

Okay, what about this? The second example here, so we have this RDD of people. And we do people.take, what happens? Where will the array of person representing first(10) and up. Well by now I think you'll guess the driver program. Because in general when you execute an action it involves communicating data back from the worker notes to the driver program. So what you do here is you're telling the executor nodes to do this computation and to send it back to the driver nodes. So running an action actually sends data back to the driver nodes. So what you're doing is you're giving an order to the worker nodes to do some work. And then the work nodes are sending their data back to the driver node. So the driver program is the one that has this array of ten person objects.

Cluster Toplogy Matters!

Moral of the story:

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

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

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

So the moral of the story is, to make effective use of RDDs, you really have to understand a little bit about how Spark works under the hood. In particular, you have to know the anatomy of a Spark Job. You have to know that there's something called the driver program, and that's sort of the brains of the application, and that a bunch of work is being farmed out to worker nodes and executors all over the place. Right, and due to this API which is mixed, eager and lazy, it's not always immediately obvious upon first glance which part of the cluster a line of code might run on. For example, transformations. So transformations are queued up and then when an action comes along, then all that staff that got queued up gets optimized by the driver program, sent to the worker nodes, and then the worker node sends something back to the driver node. So when you write a line of code like foreach you really have to think about where that code is going to execute. If you write foreach and you want to do some foreach println (不是只 println, 而是 foreach println) to see what is going wrong with your data set, you're not going to see the result of that because that's being executed on the workers. So you really gotta think about where your code is executing. It's on you to know where the code is executing. So remember, even though RDDs look like regular Scala collections upon first glance, unlike Scala collections, RDDs require you to have a good grasp of the underlying infrastructure that they're running on. That's the take away for this session.