Reduction Operations

Reduction Operations

Big Data Analysis with Scala and Spark

Heather Miller

In the last few sessions we focused on operations like map, filter and flat map and we got a lot of intuition about how these sorts of operations are distributed over a cluster. So for example, how a distributed map or how a distributed filter is done. And also along the way we got a little bit of intuition about the anatomy of a Spark job, where these operations are actually running. But so far, we haven't focused on distributed reduction operations. In this session, we're going to focus on reduction operations in Spark.

What we've seen so far

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

Spark's Programming Model

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

So far, we've defined distributed data parallelism, and we saw intuitively that Spark implements this model. And most importantly in the first few sessions I hope you got a feel for what latency really means to distributed system. Then we covered Spark's programming model. In the beginning, we learned that from 10,000 feet to really far above Spark looks like Scala collections. But internally, but doesn't behave very much like a Scala collections. In particular Spark uses laziness to save time and memory on the latency front. In that vein, we saw transformations and actions and we learned about caching and persistence. Or said another way, we learned how to selectively cache data in memory in order to save compute time. We also saw how the cluster's topology comes into the programming model. You can't forget where your code is running.

Transformations to Actions

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

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

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

So far, most of the intuitions that we've developed have focused on distributing transformations such as map, flatMap, and filter. And we spent a lot of time visualizing how transformations like map, flatMap and filter are parallelized and distributed. Though, if you notice, we didn't yet consider how actions are distributed. So think about operations like reduce. How are these actions distributed? They're very different from operations like map or filter.

Reduction Operations, Generally

First, what do we mean by "reduction operations"?

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

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

Okay, since they're so different, maybe we should first clarify what we mean by reduction operations. By reduction operations, I mean operations such as fold, reduce, or aggregate from Scala collections. All of these operations including their variance like fold left, or reduce right have something in common about how they're computed. They have some shared structure of computation. Can you remember what that is? So we can conceptually say that reduction operations walk through collections and combine neighboring elements together to produce a single combined result. Note that this is unlike map because map returns a new collection and reduction operation, essentially combines the elements of the collection to some kind of single value like an integer or a double or a string. So many of Spark's actions can be considered reduction operations, but some actions are not reduction operations. For example, saving things to a file, that's not a reduction operation. Yet it's still an action in Spark.

Reduction Operations, Generally

Reduction Operations:

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

(rather than another collection)

Example:

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

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

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

From online:

reduce(<function type>) takes a Function Type ; which takes 2 elements of RDD Element Type as argument & returns the Element of same type
Syntax
def reduce(f: (T, T) ⇒ T): T

fold() is similar to reduce except that it takes an 'Zero value'(Think of it as a kind of initial value) which will be used in the initial call on each Partition
Syntax
def fold(zeroValue: T)(op: (T, T) ⇒ T): T

The difference between fold, foldLeft, and foldRight.

The primary difference is the order in which the fold operation iterates through the collection in question. foldLeft starts on the left side—the first item—and iterates to the right; foldRight starts on the right side—the last item—and iterates to the left. fold goes in no particular order.

Fold does not go in any particular order.

So just to give you a more concrete example of what we mean by reduction operation, let's look at a simple illustration. Let's say we have the case class, Taco [LAUGH] with two fields. One named kind, which is a type String, and one named price, which is type Double. And let's say we have a list of instances of Taco and we'll name that list tacoOrder. So we have this list here of tacos. We can use a

reduction operation to calculate the cost of the tacoOrder. In this case, we'll use a fold left operation, and we'll walk sequentially through the collection of tacos, accumulating and summing up the price of all the tacos starting with the initial price of the zeros. So starting with this 0.0 double here, we're going to accumulate up the prices and sum them up. And at the very end, we'll have the price of all these tacos summed up. This is very simple, and I hope you remember these kinds of operations from the last few Scala courses. My point in showing it to you is just to remind you of the difference between this kind of operation where the goal is to combine together the elements and to return a single result. With that of the transformation like operations we saw earlier, like map or filter, where we apply some kind of function to each element in the collection and we return a new collection.

Parallel Reduction Operations

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

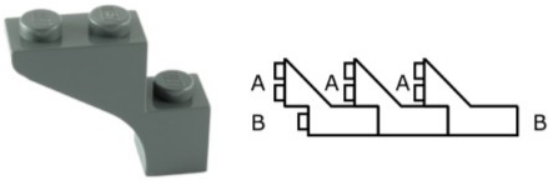
Which of these two were parallelizable?

`foldLeft` is not parallelizable.

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

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

— Scala API documentation



B 就是 sum 的類型(即 double), A 就是 taco 的類型(即 Taco)

So now that you remember what I mean by reduction operation, let's recall parallel reduction operations from the parallel programming course. In that course, we learned about foldLeft and fold. Do you remember which of these two were parallelizable? I'll give you a moment to think about it. Try to visualize the two operations and think carefully about why the pattern of computation for one might be parallelizable, and why the other one might not be. You might remember that foldLeft is not parallelizable. According to the Scala API documentation, foldLeft applies a binary operator to a start value and all elements of the collection or iterator going from left to right. So this is the start value here. Notice here we have to execute things from left to right sequentially. And if you remember from the parallel programming course, Alex gave this really cool visualization with a LEGO block which shows how having these two types that you combine down to one type so AB to B, AB to B. This forces us to have to execute things sequentially. But for me, this isn't the clearest illustration, even visually, so let's step through a simple example to see why we can't parallelize foldLeft.

Parallel Reduction Operations: FoldLeft

`foldLeft` is not parallelizable.

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

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

Concretely, given:

"1234"

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

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

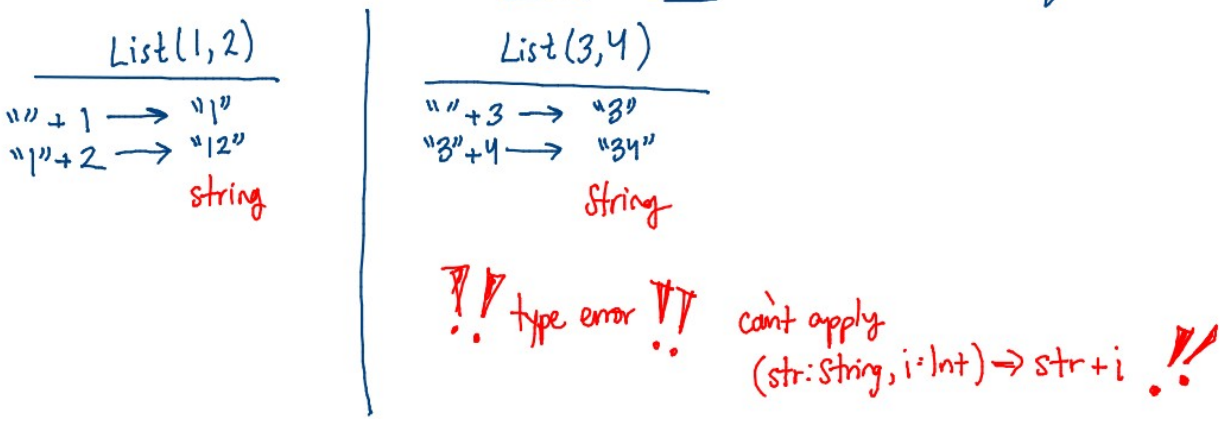
In particular, this LEGO block visualization tries to show that if we have to change the result type from A to B, we have to execute things sequentially from left to right. So let's say we have a list with four integers in it, 1, 2, 3, and 4. And now we want to use foldLeft to walk through all the elements in the list and to combine them together into a single string. So at the end, what we want is a string, 1234. So just something like this. So let's step through this simple example and let's try to paralyze it by hand to understand why the types are stopping us from doing this in parallel.

Parallel Reduction Operations: FoldLeft

foldLeft is not parallelizable.

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

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

String

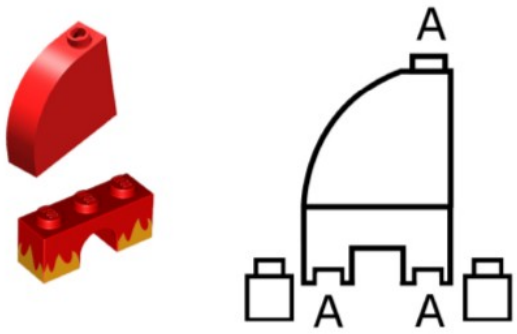


So remember that the idea in parallel programming was that we would be breaking these collections into pieces and individually working on each piece. So let's break this xs list into two pieces. And let's try to compute this foldLeft function on each individual piece in parallel and then let's combine them later. So let's start with the first element in our first chunk. We take the 0, the empty string. And then we concatenate it with the first element. This gives us 1. Then we do the same for the next element. Now we have 1, we add 2, now we have the string 12. And we can imagine that in parallel the same thing is happening here. So we start with our 0 and we take the first element which is 3, and we get the string 3. Same thing happens for the next element. We have now 3, we concatenate with 4 and now we 34. So these things can be done in parallel independently of one another. Now we have to combine them together. Notice our function is from type (String, Int) to String. But the types of these individual pieces now, if we want to try and combine them together, is (String, String). Which doesn't fit into this function any more. So now we would have a type error if we tried to combine these things. So I hope that illustrates what we mean when we say that because the type has to change we can't do this operation in parallel. Suddenly we can't combine these two individually computed pieces together any more, due to the types. So this is why foldLeft is not parallelizable.

Parallel Reduction Operations: Fold

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

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



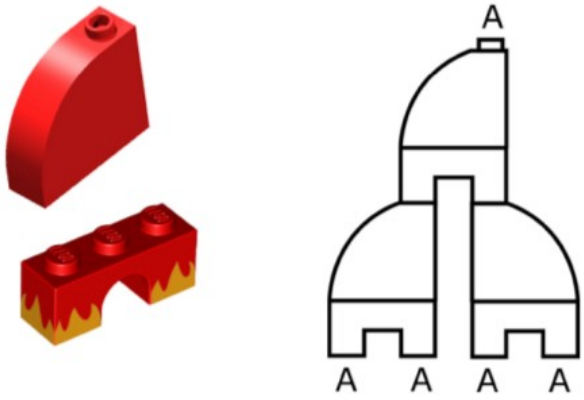
It enables us to parallelize using a single function `f` by enabling us to build parallelizable reduce trees.

So that brings us to the fold operation. And as a reminder, fold is very similar to foldLeft, except with the requirement that we always have to return the same type. So remember in the signature of foldLeft, we had B, A here, now it's A, A. Everything is the same type. And if we recall our last example, if this list here was a list of strings instead of a list of integers, then I hope you can see how, due to the types being all the same. Where everything is a string, or taking strings and combining strings or returning strings, then it's easy to parallelize these operations. So said another way, the fold function enables us to parallelize things using a single function as a parameter, where the types are all the same and by enabling us to build parallelizable reduce trees.

Parallel Reduction Operations: Fold

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

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



And if the LEGO visualization helps, here's how it would look to build one of these reduced trees. Here you can see how the tree could be broken up and worked on independently by different workers. So you can have somebody working on this part of the tree and somebody working on this part of the tree totally independently of one another. And then they can still combine all of the results together. So I hope you have some intuition now for the differences between fold and foldLeft and why it's easy to parallelize one operation but not easy to parallelize the other one.

Parallel Reduction Operations: Aggregate

Does anyone remember what `aggregate` does?

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

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

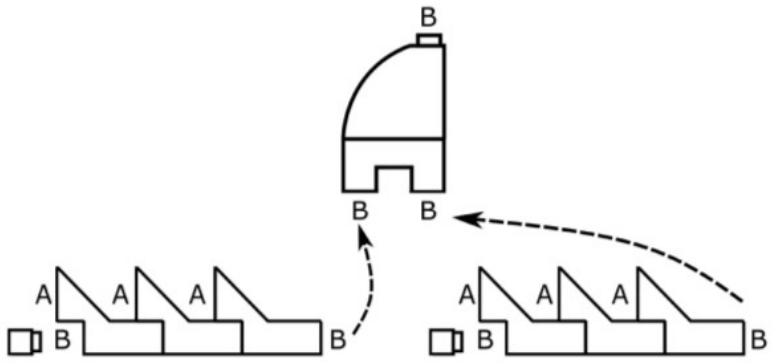
Properties of `aggregate`

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

And that brings me to the aggregate operation. Does anybody remember what aggregate does? This is important, so I'll give you a second to try and remember. I'll give you a hint, though. As it sounds, it's similar to foldLeft or fold. However, the signature is a little different. Okay, so here's the signature of the aggregate method for regular scala collections. Now we have three parameters in two parameter lists. Like before, we have a start value of type B, but now we have two functions instead of one. We have seqop and combop. The first one, seqop, represents a sequential operator, and like in foldLeft, it operates on two types. The second one is called combop, And it represents a combination operator. And like in regular fold, it only operates on one type, in this case B. So this is kind of like the function in foldLeft and this is kind of like the function in regular fold. While the signature might seem complicated, it's actually great for parallelizing things over a cluster. In fact, it's considered to be even more general than fold or foldLeft because it's both parallelizabl and it makes it possible to change the return type to something else.

Parallel Reduction Operations: Aggregate

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



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

So to just bring back the cool LEGO visualization, we can now visualize how it's possible to parallelize the aggregate operation. As you can see, what this does is it allows us to do the work of the foldLeft on separate chunks in parallels. So you can have individual workers doing this work. So think individual nodes or individual executors doing this work. Then we can change the types using this foldLeft sequential style reduction operator. But we can do it in parallel line chunks. And now that we have this combop function here, it makes it possible to build these nice reduced trees that are so easy to parallelize.

Reduction Operations on RDDs

Scala collections:
fold
foldLeft/foldRight
reduce
aggregate

Spark:
fold
~~foldLeft/foldRight~~
reduce
aggregate

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

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

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

Okay, great. Now that we have an intuition about why some operations can be done in parallel and why some can't be, and how to do everything we want to do like changing types and doing reduction computations in parallel. Though how does this relate to RDDs? How do we distribute these things? Well, let's start by looking at which of these reduction operations exist on regular Scala collections and on Spark. Of course, what we see here is that one operation we knew couldn't be parallelized simply doesn't even exist on RDDs. So sequential foldLeft and foldRight operators are simply not defined on RDDs at all. You have no choice to use them. This could be perhaps a little frustrating because foldLeft tends to be a pretty commonly used reduction operator when working with collections. It seems to be one of the more popular variants, and now you suddenly don't have access to it all. So what does this mean? Well, since in regular Scala collections our only choice for changing the return type of reduction operation was to use either the foldLeft or foldRight operation or aggregate. This means that now In Spark, since there is no foldLeft or foldRight operation, we can only use the aggregate operation if you have to change the return type of your reduction operation. So now you have no choice. You can't reach for this familiar foldLeft anymore. You have to use this weird aggregate thing that maybe you didn't use so much before in regular Scala collections. Though but you might be saying, hey, hold on a second. Why not still have a serial foldLeft or foldRight operation in Spark? It would mean a simpler and more convenient API for users to use, and it would be more in line with regular Scala collections, right? Well, it turns out that trying to do anything serially across a cluster is actually very difficult. Enforcing ordering in a distributed system is very hard and sometimes impossible. And in any case, it requires a lot of communication and synchronization between nodes, which we have learned throughout several of our sessions now is extremely costly. So it simply doesn't make sense to try and make sure that something happens before another thing in a distributed system. Which means it typically doesn't make a lot of sense to try and make available serial operations, like foldLeft on a cluster.

RDD Reduction Operations: Aggregate

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

Example:

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

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

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

In fact, I'll even argue that in Spark, the aggregate operation is actually a more desirable reduction operation a lot of the time. Why do you think that's the case? So remember, we're in distributed thinking now. We have big data sets. What is it about the jobs that you might want to do with Spark that would make an operation like aggregate so desirable to use? Think about it. As you'll get to know from the programming assignments and hopefully from real data analysis jobs. As you'll get to know from the programming assignments, sorry. And hopefully from real data analysis jobs you may one day do on real clusters with Spark, much of the time, when you have some large amounts of data, you typically have very complicated records or elements of your RDD. So the individual element types are very big, nested, complex things. They're often not simple integers or strings like we've been looking at. But instead, they're a really complex data type. And often, you don't need the whole thing to do the analysis that you may want to do. You may want to project down this really complicated data type into something simpler to actually work on. For example, if you wanted to do a computation involving the data stored on Wikipedia, you may have a data type called WikipediaPage which contains all kinds of fields. Probably more than what's listed here, but which could include things like the title of the article, the redirectTitle of the article, the timestamp of the article, the username of the last person to contribute to the article, and the actual text contained within the article. So you can also imagine many more possible fields, like a list of articles linked to in that Wikipedia article, etc. And you can image that to do some kind of interesting computation or some kind of interesting analysis on your Wikipedia data, you may only care about the title of the article and the timestamp. In this scenario if you had to keep the full text of every Wikipedia article around in every element of your RDD, it could waste a lot of time and a lot of memory to carry around this perhaps very big full text string around in your accumulator. This is why you might find that you would use the aggregate operation more often in Spark then you did in Scala collections. Because typically you want to get rid of stuff that you don't need and project down to a smaller data type.

Pair RDDs

ÉCOLE POLYTECHNIQUE
FÉDÉRALE DE LAUSANNE

Distributed Key-Value Pairs (Pair RDDs)

Big Data Analysis with Scala and Spark

Heather Miller

In this session, we're going to focus on distributed key values which are a popular way of representing and organizing large quantities of data in practice. And in Spark, we called these distributed key-value pairs "pair RDDs".

Distributed Key-Value Pairs

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

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

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

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

If we think back to the functional programming in Scala courses, when we were focusing only on sequential Scala in the basics of functional programming, we didn't work with any data types called key value pairs though, we did work with a collection type called a map. And if you recall, maps are essentially collections of keys and values. So while we didn't call them map key-value pair,

conceptually, it's the same thing and this concept goes by other names in other languages, such as associative arrays or dictionaries in languages like JavaScript or Python. Though if you think back to the single machine scenario even though most languages have something like a map or a dictionary as a core data structure, maps, dictionaries, et cetera maybe weren't the most often reached for data structure on the shelf of data structures that you had to choose from when you were doing these programming exercises in the earlier functional programming courses. You might have used other data structures like lists or even arrays more often than you might have reached for something like a map. However, in the world of large scale data processing and distributed computing, the opposite is actually true. It's actually more common to operate on data in the form of key-value pairs than anything else. In fact, when google designed MapReduce, focusing on all data as key-value pairs was a key design decision based on real use cases involving large amounts of data at Google.

Distributed Key-Value Pairs

(2004 research paper)

MapReduce: Simplified Data Processing on Large Clusters

Jeffrey Dean and Sanjay Ghemawat

jeff@google.com, sanjay@google.com

Google, Inc.

Abstract

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

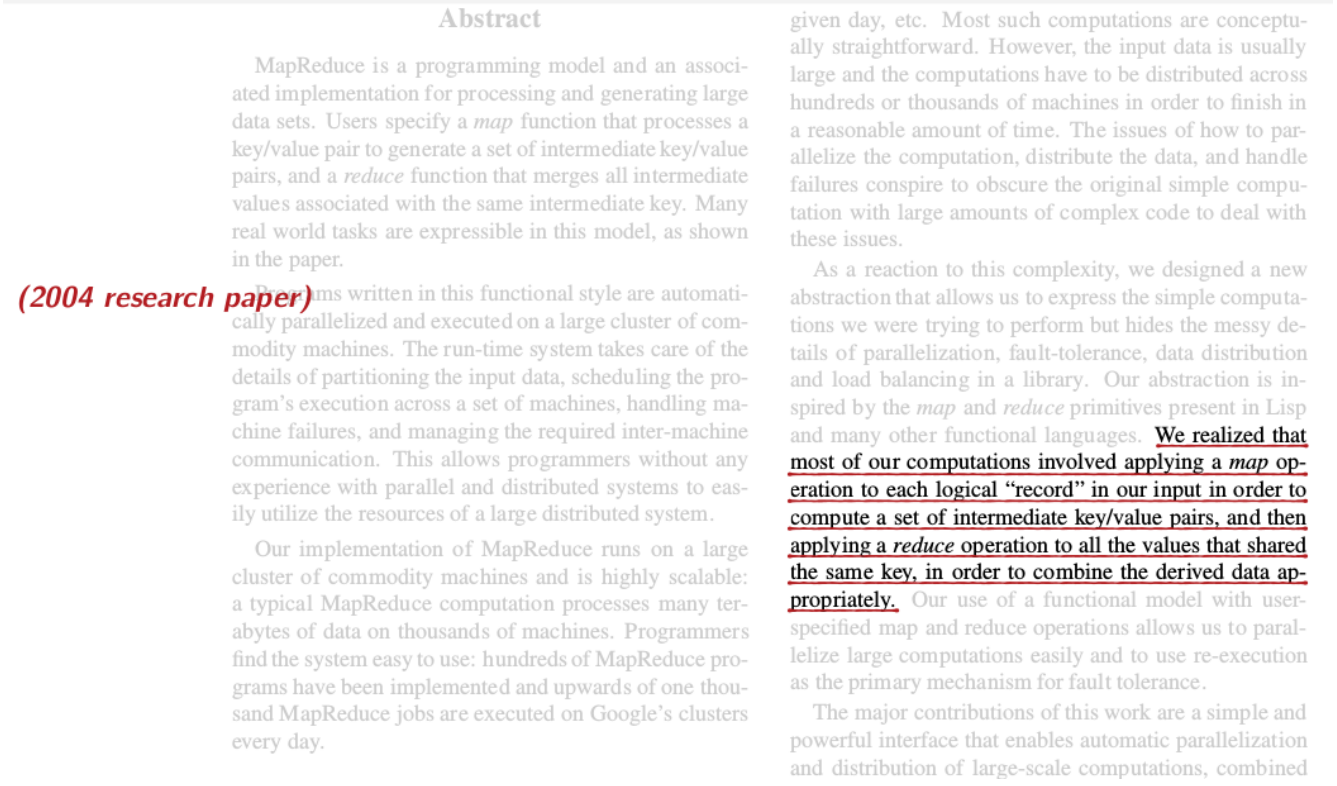
Programs written in this functional style are automatically parallelized and executed on a large cluster of commodity machines. The run-time system takes care of the

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

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

In fact, the original designers of MapReduce even explained the rationale surrounding their focus on key-value pairs very clearly in their original research paper, which present MapReduce to the world back in 2004.

Distributed Key-Value Pairs



Abstract

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

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

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

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

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

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

(2004 research paper)

They said here, we realized that most of our computations involved applying a map operation to each logical record in our input in order to compute a set of intermediate key value pairs and then applying a reduce operation to all of the values that shared the same key in order to combine the derived data appropriately. Or said another way, computations that they were already doing at Google always ended up producing key-value pairs. So, they focused the design of MapReduce around this common pattern for manipulating large amounts of data.

Distributed Key-Value Pairs (Pair RDDs)

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

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

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

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

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

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

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

This is due in large part to the fact that large datasets are often made up of unfathomably(深不可测) large numbers of complex nested data records. So, complex elements of a large dataset like the Wikipedia case class we saw in the previous session. And of course, the case class that I showed you was just an example and a simplification of such a record. Given such complex records and the need to do computations on them, often data analysts need to project down these complex data types into key-value pairs in order to operate on them. Here's another example of a record. A single element in this case and it's in JSON, but this is the shape of perhaps a single element that you might find in an RDD. The intuition I want you to get here is that often, there are many fields and there are lots of rich nested structure to these records. There can be objects nested in objects nested in objects, and it can go quite deep. So let's say, we have something like this JSON record on the left as our input data and the analysis that I want to do may focus only on the cities that certain properties are in. So here, we have some properties and I care about the cities and perhaps the street addresses, but maybe I want to do some analysis and I don't care about the rest of this data. I only care about this part of the dataset here. So in this case, it may be desirable to create an RDD of properties of type pair, string, property where the key of type string represents the city that a property is in and the property type here represents the corresponding value like we see in the record to the left. We have the case class four here where the property contains a street of type string, a city of type string and a state of type string. So you can imagine that given bunch of this properties, it could be desirable to group them by their cities as the key to do computations then on this groupings of properties by city. This is one scenario where on a large dataset manipulating your data into a key-value pair might be desirable.

Distributed Key-Value Pairs (Pair RDDs)

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

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

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

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

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

As alluded to earlier in the session in Spark, we call these distributed key-value pairs Pair RDDs and they are particularly useful, because they allow you to do special operations per key in parallel and they allow you to group data by keys across the network. Very importantly, Pair RDDs come with special methods for working with data that's associated with a corresponding key. That is Pair RDDs have extra methods related to keys that regular RDDs don't have. So if you try to call some of these methods on an RDD which is parameterized by type integers, so an RDD full of integers, the compiler would say, nope, I can't do it and will give your error saying, the method cannot be found. So remember, if you have an RDD with a type parameter that is a tuple or pair, it is treated specially by Spark. It has special extra methods.

Pair RDDs (Key-Value Pairs)

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

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

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

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

And just to give a sense of a handful of these special methods that we have on Pair RDDs, some of the more commonly used ones are groupByKey(), reducedByKey and join. We'll cover these methods in a lot more detail very shortly.

Pair RDDs (Key-Value Pairs)

Creating a Pair RDD

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

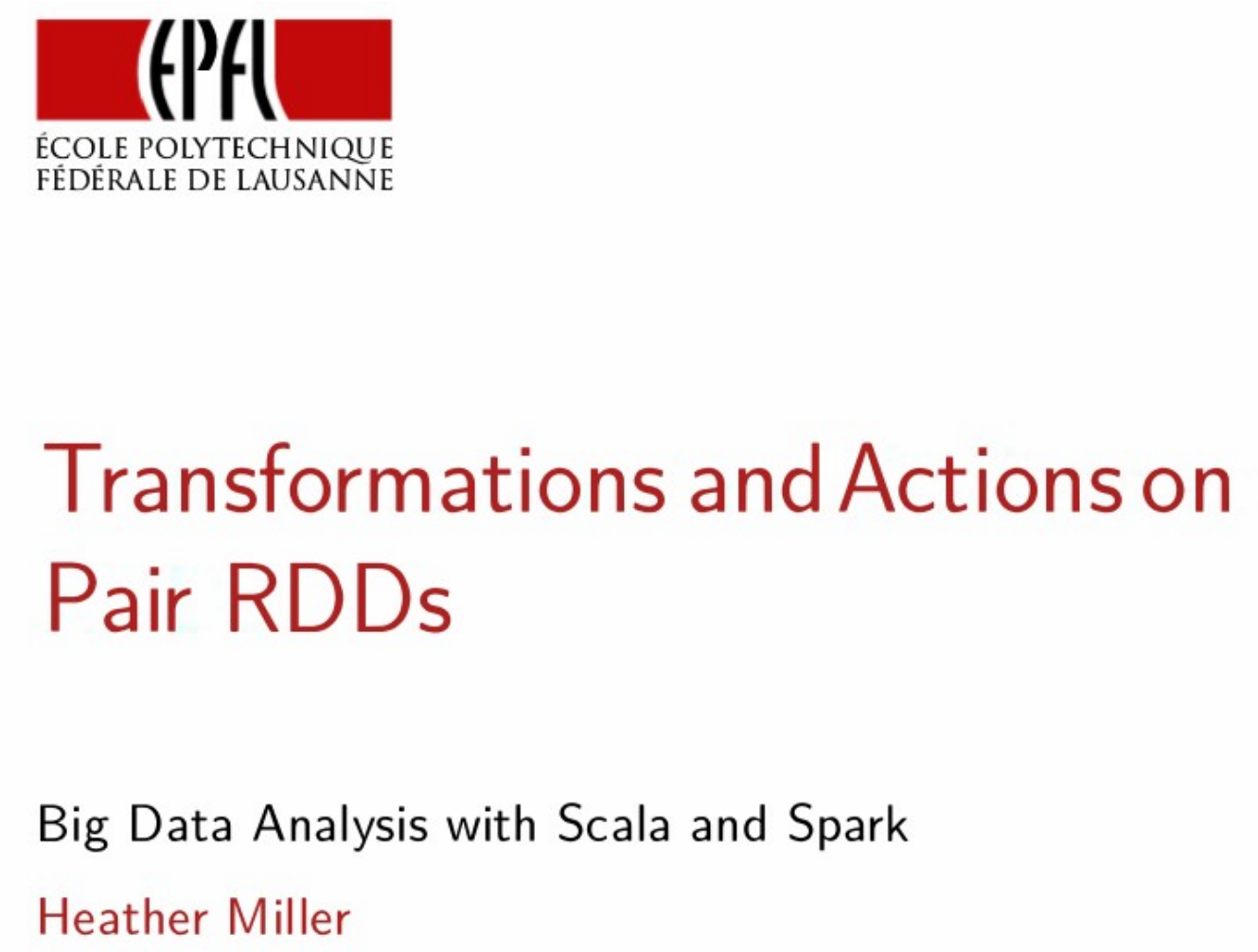
```
val rdd: RDD[WikipediaPage] = ...

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

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

But first, you might ask, well, great, how do I actually created a Pair RDD? It's actually quite simple. Pair RDDs are most often created from already-existing RDDs using operations such as the map

operation on RDDs. So that said, let's do a quick quiz. Let's say, I wanted to create a Pair RDD from an RDD full of these Wikipedia pages that we saw in a previous slide such that the key of the Pair RDD represents the title of the Wikipedia article and the value of the Pair RDD represents the text of the Wikipedia article. What method would I have to call with which arguments to create this Pair RDD from this val RDD above? So again, as a hint, you can use some kind of transformation operation to then create a Pair RDD. What would you use? It's simple. In this case, I can just call map on the original RDD here and I pass a function here which selects the title and the text from the Wikipedia article and then makes them into an element of a pair. So, this is now a Pair RDD and it has all of these special key-value pair transformation methods available to you to be used now. So now, the type is what we saw in the previous slide. The type looks something like this now, so it gets all the special methods. So now, you can do things like reduceByKey or join on this new pairRdd that you couldn't do before on this Wikipedia rdd. In the next session, we're going to dive into a lot more detail about these operations on Pair RDDs. So, these and more.

Transformations and Actions on Pair RDDs



Now that we've seen pair RDDs, in this session we're going to focus on some of the most commonly used pair RDD operations that you're going to find in the wild.

Some interesting Pair RDDs operations

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

Transformations

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

Action

- `countByKey`

Just like we saw before for regular RDDs, these common operations can be broken into two categories, transformations and actions. There are several special transformations specific to pair RDDs that we'll cover, and one special pair RDD action that's called count by key. Now if we look at the some transformations that we'll cover, many look familiar. And functional programming in scholar course, remember operations like groupBy for example. So there's groupBy key that seems kind of similar. And of course, remember stuff like reduce. MapValues, this guy here, sounds kind of like map. We might be familiar with some idea of what join is from some experience that we've had with databases in the past. So these guys kind of sound familiar as well.

Pair RDD Transformation: groupBy~~Key~~

Recall groupBy from Scala collections.

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

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

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

注意上圖講的是 Scala 中的 groupBy, 而不是 Spark 中的 groupByKey. 上圖的不好懂, 看下圖的例子就明白了.

But let's drill a little bit deeper into some of these operations. Let's start with groupByKey. To do that, let's first recall groupBy from regular Scala collections. Do you remember what this operation does? So just remembering groupBy from regular Scala collections. If we look at the API documentation, groupBy has the following signature. groupBy takes a function from A to K, so a function from some type to the type of the key that the groupBy should return. And of course it returns a map, where the key maps to a collection of elements of type A. The EPI docs succinctly describes the semantics of this operation as follows. So, groupBy partitions this traversable collection into a map of traversable collections according to some discriminator function. We can also say here's a little more simply by looking at it the following way. So assuming you have some kind of collection, the groupBy operation will break up that collection into two or more collections themselves, according to some function that you passed a groupBy. So, according to this function here. The results of the function are the key. So that if you apply this function to individual elements of the original collection, whatever the result is of that function application is the key that that element should correspond to in the resulting map returned by the groupBy operation. So the map returned by this groupBy method contains keys mapped to collections of those key corresponding values. It still sounds a little confusing, doesn't it? I'm trying to make it simpler, but I don't know if you can visualize it.

Pair RDD Transformation: groupBy~~Key~~

Recall groupBy from Scala collections.

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

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

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

So let's make this a little bit clearer by looking at an example for regular Scala collections. Let's assume that we have a list of integers here, it's called ages. And these integers represent the ages of people. Now let's group this listed ages into categories. One for child. One for adults. And one for seniors. So we have three categories. That means we need a function that when it's applied to the integers in this list, it determines which key that age should correspond to. So let's do a simple conditional express. If the age is greater than or equal to 18 and less than 65, then we know that this person should be an adult. They have to be an adult. Else, if this person is under 18 years old, then this person is a child. And finally, anything else, we say this person is a senior. So we assume that they're over 65. And that means that they're a senior. So now what groupBy does here is applies this function to each element in the ages list here. And it determines the string that corresponds to the age, whether it's a child, adult, or a senior. And then it produces a map as a result, with a list of the ages that correspond to each key. So now we have a map containing three keys, senior, adult, and child, where the corresponding values are lists of the ages that we found in our original list of ages that fall into those categories, that are grouped into those categories. Hence, why we call the operation groupBy. We're grouping elements into a category based on some discriminating function. I hope that makes the groupBy operation a little clearer now.

Pair RDD Transformation: groupByKey

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

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

Example:

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

val groupedRdd = eventsRdd.groupByKey()
```

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

So let's get back to Spark now. Spark's groupByKey operation can be thought of as groupBy on Pair RDDs, but it's specialized on grouping all of the values that have the same key. So that means we're doing it on a collection that's already a collection of pairs, or in this case an RDD of pairs. A pair RDD. That means that we don't have the discriminating function anymore. Now, groupByKey has no arguments. In this case, we only want to group things by the keys that they already have. So we just want to take a RDD that's full of many key value pairs, and we want to now return a RDD with all of the values collected that correspond to a specific key. All of these collected values should be put into some kid of iterable for some kind of regular Scala collection. So we go from countless little pairs of keys and values with possibly many of the same keys floating around with different values. And now we have just one key that matched to a collection of all the values that correspond to that key. And then they have been grouped into a regular Scala collection. To make that a bit more concrete, let's look at an example. Let's say we have a case class called Events that represents events, here. And each event should have an organizer. So let's just say there's a concert or something coming up. And somebody is organizing it. The concert will have a name. And that's the string here. And each event should have a budget of some kind. And so, our first step is to make a pair RDD. So, we do that with this map function here. And we organize things such that our key is the event organizer's name. And then, the value is the budget for the given event. Now, we call by key on that new pair RDD we just created on the previous line. So, what does this call do? I'll let you think about it for a moment. Remember, the key is the organizer.

Pair RDD Transformation: groupByKey

Example:

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

val groupedRdd = eventsRdd.groupByKey()

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

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

Haha, it was a trick question. The answer is that it does nothing as it is, it returns a reference to an unevaluated RDD. Remember, that groupByKey is a transformation. It's a transformation, so that means it's lazy and nothing happens when you call it. It returns a reference to an unevaluated RDD. But okay, if we called an action on it, in this case collect, here, now we have an array that we've gotten back from our RDD, which forces that computation to take place. Then we can call for each on that array and then print line to just print the elements of the array returned by the collect to see what the result looks like. Assuming that we have a few events that have been organized by the same organizer, the result will look like this here. The organization Prime Sound, for example, has just one event with a budget of $42,000 or Francs or whatever currency it's in. And then the other organization called Sport Org seems to be organizing three different events, each with budgets of $23,000, or Franks, $12,000 of Franks, and $1400 of Franks. So that's group by key, and it's actually quite an often reached for method on large data sets. You might find yourself deciding, my gosh, it would be really useful to group these things by key and then do something with these grouped elements.

Pair RDD Transformation: reduceByKey

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

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

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

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

val budgetsRdd = ...
```

However, there's another important transformation operation on a pair of RDDs, called reduceByKey. Conceptually, you can think of reduceByKey as two operations. First, the groupByKey that we saw in the previous slide, and then reducing all of the values in the collection that corresponds to each key, so the resulting collections that come back. Then we reduce on those that correspond to each key, via some function. And what's really, really important to note here is that reduceByKey is actually a lot more efficient than doing groupByKey and then reduce independently. We'll see why shortly. But this is an important little note that you should hold on to. So let's look at the signature of reduceByKey. As we can see, reduceByKey takes a function which only cares about the values of the paired RDD. So we're using V here to represent the values. So we don't actually do anything with the keys in this function, we only operate on the values. This is because we conceptually assume that somehow the values are already grouped by key and now we apply this function to reduce over those values that are in a collection. So this function, we imagine it reducing on all of the elements in the compact buffers here.

Pair RDD Transformation: reduceByKey

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

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

val budgetsRdd = eventsRdd.reduceByKey(_+_)

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

Again, to make this more concrete, let's look at an example. Let's reuse the events pair RDD from the last example. Now let's say we would like to calculate the total budget per organizer over all of their organized events. Now that we have a little bit of intuition for what reduceByKey does, how might we use it to calculate the total budget per organizer given a pair RDD with the key is the organizer and the value is the budget of some event? I'll give you a minute to figure out how to implement budgetsRDD on your own. This should be quite easy, we just call reduceByKey on the eventsRDD and we pass into it a function lateral that sums into the adjacent values. Again, remember that reduceByKey is a transformation, so that means it's lazy and nothing happens after you call it. Even though it looks kind of like a reduce, which if you recall, is an action on regular RDDs. But on on pair RDD is reduceByKey is a transformation. So that means we have to call some kind of action after our call to reduceByKey. In this case, again, we call collect to start the computation. And then we can print out the results with the regular foreach on the returned array. And here what we can see in the results are that there are four organizations organizing events. And the total budget across each organization is shown as the second element in the returned pairs here. So I hope that illustrates clearly reduceByKey for you, and it's pretty important, because you're going to find this useful in your programming assignments as well.

Pair RDD Transformation: mapValues and Action: countByKey

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

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

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

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

Let's look at another kind of transformation called mapValues, and the one action I mentioned at the beginning of this session, called countByKey. So let's start with mapValues. The mapValues transformation, as its name suggests, applies a given function only to the values of a key value pair. This is something often you may want to do when you want to do a map on a pair RDD, but you remember you have to somehow handle the key in function argument that you pass to the map. So you can think of this operation as a short hand for skipping over having to deal with the keys. It just is a little bit shorter in this example. It has some other benefits as well, we'll talk about that in a later lecture. So said simply mapValues applies functions only to the values in the pair RDD and it skips over the keys. It's pretty straightforward in its way people typically do mapping operations on pair RDD's, they don't really use the map operation a lot of the time. The other operation that we'll look at is countByKey which is an action. Like it's name, it simply counts the number of elements per key in a pair RDD. Importantly, it returns a regular Scala Map, so not anything distributed, but like a regular Scala Collections Map. And that maps the keys to the corresponding counts so you then have a count per key of all of the values that you maybe have per key. And it's very simple and remember it's an action just like the count operation regular RDDs but it's specialized in this case for pair RDDs.

Pair RDD Transformation: mapValues and Action: countByKey

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

```
// Calculate a pair (as a key's value) containing (budget, #events)
val intermediate =
  eventsRdd.mapValues(b => (b, 1))
           .reduceByKey(
```

(org, budget) → (org, (budget, 1))
K V

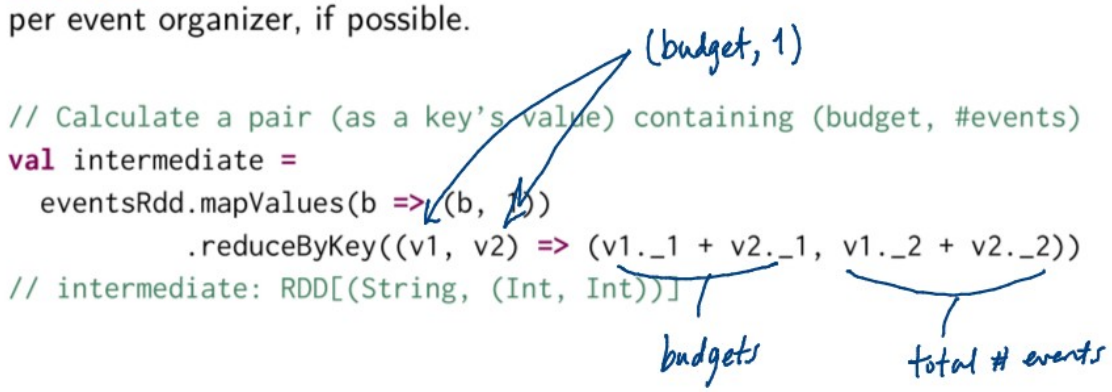
Result should look like:

(org, (totalBudget, total# events organized))

As usual, let's look at an example to make these operations a bit more concrete. Again, let's use the event pair id already used in previous slides, and instead let's use each of these operations if we can to compute the average budget per organizer. So now we are taking an average. I'm going to help you a little bit in the beginning, so lets do this step by step, but, let's start with an intermediate RDD here. Lets calculate a pair as the corresponding value of the organizer key. That means our value is also a pair, which will represent the total budget, and the total number of events that correspond to that event organizer. So this is the value here, the value is a pair of its self. And it has the first element is the budget, the first element is the number of your events that correspond to that organizer. Okay, so first question. Can we use countByKey here? Let's start with mapValues. Remember, our pair RDD called eventsRDD contains key value pairs where the key is the organizer and the value is the budget for some event. We know we need to take a sum of the number of events, so we can use mapValues to make a new pair from our existing key value pair that makes it easier to sum up the events per organizer. So that later on, we're going to want to sum up the events. So for the value and the key value pair, we make a new pair where we just leave the value as it was whatever the value was before and we have a second element of the resulting pair, which is just the number one. Just to write this out, in the eventsRDD, we have. The organizer as our key and the budget as our value. This is before the map values. And now after the map values, we're going to have is this. Since we're using mapValues, the key stays the same, nothing happens to it, and the value changes. So basically what we're doing here this b is at the budgets, so we say okay, we keep the budget, whatever, and we add a 1 as well. So do budget and an integer 1. And this is now the result of the mapValues call. Now let's use reduceByKey to return a pair RDD whose value is another pair representing an organization's total budget. And the total number of events that this organization is organizing. So now, what we want is we want this here, this budget and events. We want this to be the result that we get back from this reduced by key. And this is the data that we have to start with. So our data's in this shape right now. So what do I have to pass to reduceByKey here to make sure that the result ends up in this shape. Try it for yourself.

Pair RDD Transformation: mapValues and Action: countByKey

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



So remember that the function passed to reduceByKey operates only on the values of the pair RDD, right? So let's remember the signature of reduceByKey, there was a function that was on the values only, no keys were involved. And the value, in this case, is a pair, do you remember? So this, each of these v's are pairs and these pairs. Are representing a budget as the first element and then just the number 1 as the second element, both of these. So that means in the result here, we can take the two adjacent elements which are each pairs themselves, and we just have to sum up the first elements of the pairs with the first elements of the adjacent ones, and the second elements with the second elements. So basically we're summing up budgets with budgets, and the number ones with with number ones. And then the result type should be another pair RDD where the key remains the organizer string. And the value is a pair of integers that represents the total budget and the total number of events organized by that organization. Do you see what we've done? Both operations that we called in our pair RDD managed to focus only on the values of our pairs, mapValues and reduceByKey. In both cases we never had to worry about messing around with the keys, we could just focus on the values associated with those keys. So does this make sense? Here we sum up the budgets, here we sum up 1s for the total number of events. Okay, but we're not done yet. Remember, we said we wanted to compute the average budget per event organizer over all of their events.

Pair RDD Transformation: mapValues and Action: countByKey

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

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

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

So now we finally have the information that we need to compute this average, so then I leave it to you again. Given this RDD that's called intermediate now, we're going to use this guy, how would you compute the average budgets per organizer? So we need the average budget per event organizer now. What operations would you use? Again, try it for yourself first. So here, I've used mapValues again. Remember, mapValues focuses only on the values and not on the keys when it applies its mapping function. So, I can use it to compute the average budget per organizer by simply dividing the total budget by the number of events organized by each organizer. That should now give me a pair RDD where the key still corresponds to the name of the organizer, but now the value is an integer representing the average budget per event of that organizer. And of course, just like before, I just call collect and foreach again to materialize my RDDs and kick off the computation, and then now I can see the results. But wait, didn't I ask you to think about using countByKey? Why didn't I use it here? Well, the answer is pretty simple. In all cases so far, I've been carrying around as my value another pair. So, I've had a pair as a value and I wanted to do something more complicated than simply counting each time, I wanted to compute an average. So in this case, it was easier for me to use mapValues and reduceByKey to get the input that I wanted. It doesn't mean that it's not possible to do this somehow via accountByKey, it's just that in this case I chose mapValues and reduceByKey to get it done.

Pair RDD Transformation: keys

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

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

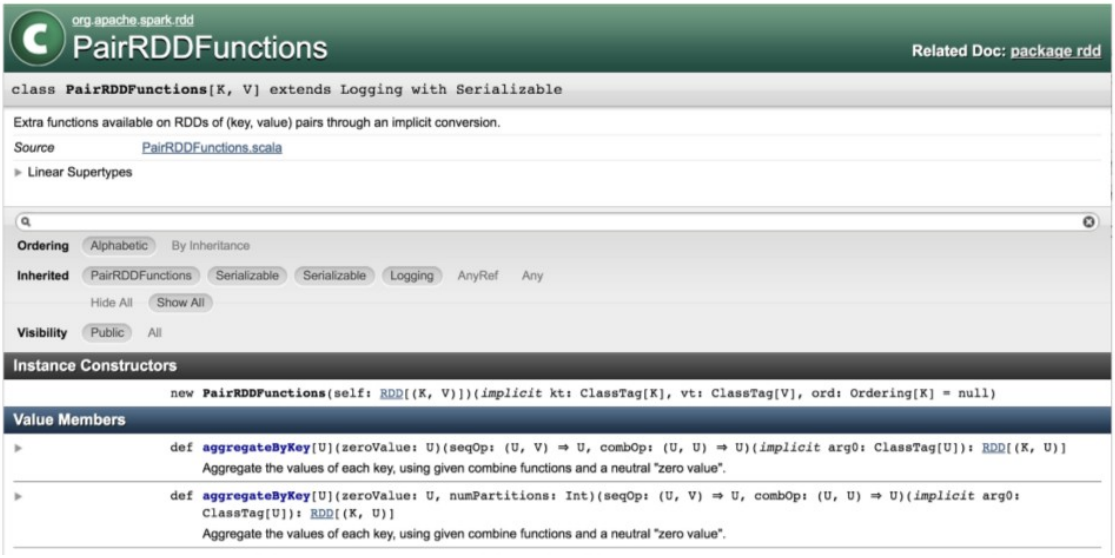
Example: we can count the number of unique visitors to a website using the keys transformation.

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

Let's move on to a different transformation called keys. The signature of keys looks like this. It takes no arguments and returns a new RDD, which represents only the keys of the Pair RDD on which it was called. Now, remember this is a transformation which means that it's lazy. At this point you might ask, well, why is that the case because operations like collect are not? They're actions and they return an array back to the master node. In this case, keys is a transformation because the number of keys in a pair RDD could be huge. So, if we tried to return all the keys to the master node like we do for collect, we could find ourselves in the situation where we overwhelm the master node and we run out of memory because we simply weren't aware that there were so many keys. This is why it's a transformation. We don't want to accidentally return an entire RDD to our master node where it doesn't fit into memory. As usual, let's look at an example to try and make the keys method a little more concrete. In this example, we're going to try and use this method to count the number of unique visitors to a website. So, let's say we have a case class visitor here and we have various pieces of information. In this case class, we have the IP address of the person who visited the site, the timestamp that they visited at and the duration that they visited for. So, let's say we have an RDD full of instances of this visitor type. And then here, I make a pair RDD out of it, the slides were missing a line. So, here we have for each visitor, we create a pair where the keys, the IP address and the value is the duration. So, what methods do we call to get a number that represent the number of unique visitors? Well, we simply have to call keys.distinct and then we have to count them up. And note that the action here is the count method because both the keys and distinct methods are transformations. So, this is how we kick off the computation here when we call count.

PairRDDFunctions

For a list of all available specialized Pair RDD operations, see the Spark API page for `PairRDDFunctions` (ScalaDoc):
http://spark.apache.org/docs/latest/api/scala/index.html#org.apache.spark.rdd.PairRDDFunctions



So, I've shown you some of the more common operations on pair RDDs. In the next session, we'll dive into joins, but I should note that other than what I've shown you in this session and other than these join operations that you'll see very soon, there are many more operations you need to pair RDDs. To see all the available methods that you can possibly call on PairRDDs, you can visit the PairRDDFunctions page. Right here, this is what it is called. It's a class in the Sparks API Documentation. This link should work for quite some time, but if for some reason the URLs get changed on the Spark website, all you gotta do is remember that this is a class called PairRDDFunctions. And all of the methods inside of it are the ones that you're going to find available to you on pair RDDs.

Joins

Joins

Big Data Analysis with Scala and Spark

Heather Miller

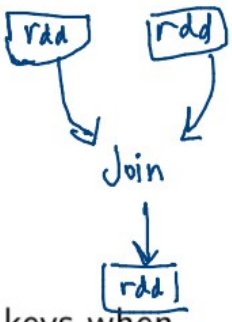
In this session, we're going to be focusing on joins. Just like in the previous session, the join operations that we'll be focusing on are unique to Pair RDDs. So that means you can't call the join methods that we show you on regular RDDs. It just won't work.

Joins

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

There are two kinds of joins:

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



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

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

You may already be familiar with the concept of joins from databases. And indeed, the joins found on Pair RDDs are conceptually similar to those that you might have come across in databases. We're going to focus on joins in Spark in this session, which are amongst the most commonly used operations on Pair RDDS. So that said, what's a join? It's pretty simple. It's just an operation that will combine two different Pair RDDs into one Pair RDD. So visually it's something like this. You have two RDDs, That's one, and that's two. And you want to combine them together somehow into a resulting RDD. And the way that you do that is using the join operation. There are two main kinds of joins. So on one hand we have inner joins, which is called just regular join in Spark. So the method just called join refers to an inner join. And the other kind is outer joins, which have two variants. So the method names in Spark LeftOuterJoin and rightOuterJoin. The difference between two variants, inner joins and outer joins, only has to do is what happens to certain elements based on their keys. Depending on whether both RDDs in the join contain a certain key or not. Said another way, when we try to join together the two Pair RDDs containing different customer IDs as the key for example, the difference between the inner and the outer join is what happens to the customers whose IDs don't exist in both RDDs.

Example Dataset...

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

Let's assume the following concrete data:

```
val as = List((101, ("Ruetli", AG)), (102, ("Brelaz", DemiTarif)),
              (103, ("Gress", DemiTarifVisa)), (104, ("Schatten", DemiTarif)))
val abos = sc.parallelize(as)

val ls = List((101, "Bern"), (101, "Thun"), (102, "Lausanne"), (102, "Geneve"),
              (102, "Nyon"), (103, "Zurich"), (103, "St-Gallen"), (103, "Chur"))
vals locations = sc.parallelize(ls)
```

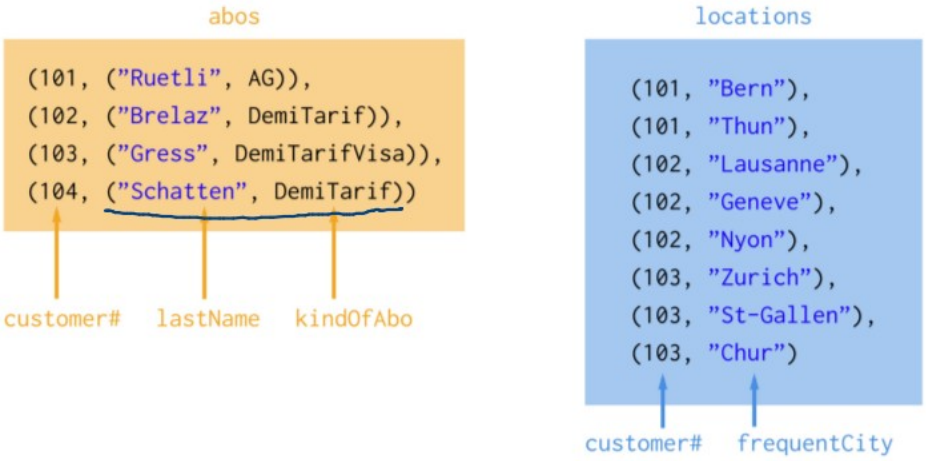
abos 格式: (Customer ID, (Customer name, card type))
locatoins 格式: (Customer ID, city)

Of course, it's easier to look at an example to better understand this difference. In this session, all of our examples are going to focus on the Swiss Rail company. So Switzerland has a very famous train system, and the company behind it is referred to as the CFF in the French-speaking part of Switzerland. So we're going to focus on examples as if we are the CFF. And we're trying to make decisions about our train service based on the travel habits of our regular customers. Rail customers in Switzerland and in other European countries almost always have some kind of discount card for traveling on a train. So in Switzerland, most of the country has some kind of discount card. They pay yearly for it. For example, the card called here, DemiTarif, gives a 50% discount on train fares. And there's also another one called abonnement general. In this case, AG we call it, you pay for it, and you get a free pass to take any public transit. So in this example, let's say we have two datasets collected by the CFF. This dataset is called abos, which stands for abonnement. It just means subscriptions. So one dataset represents customers and their subscriptions. So this should be the customers and their subscriptions here, called abos. And the other one represents the customers and the cities that they most frequently travel to. So this one called locations here. And in this one, you could imagine that it's perhaps collected from the smartphone app that people use to buy their train tickets on the go. So this is locations. It's a dataset of customers and their most frequently traveled city. And this one is abos, and it's information about the customers' subscriptions. So in this case, this customer's last name is Gress. And he or she has a DemiTarif, so a half-fare card credit card. Note that both of these datasets are Pair RDDs. So I've taken a list of pairs, and I've created an RDD out of it. This produces a Pair RDD in both cases. So these are both Pair RDDs, abos and locations. And an important thing to note is that it's possible for a customer to be in both datasets. So we can see here that this customer ID 101 is in both datasets. So this customer who has this AG card frequently travels to, for example, Bern.

Example Dataset... (2)

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

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

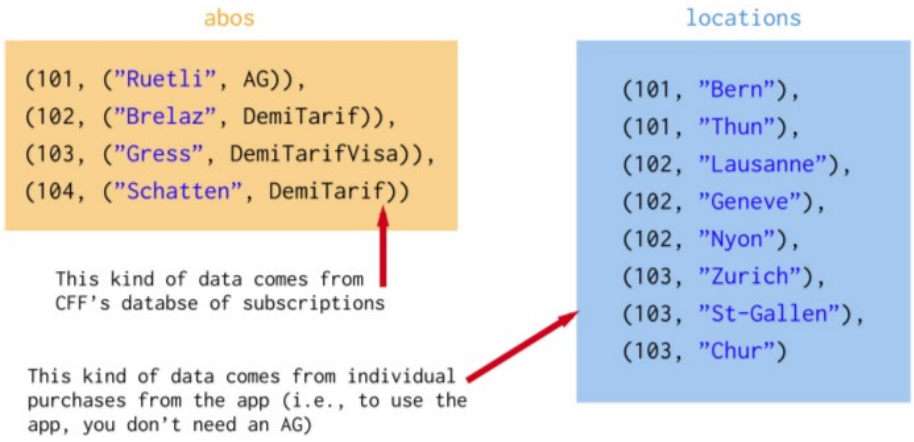


Here's a little visualization of the same dataset. So in the abos dataset, we have key value pairs where the values are pairs themselves. So these are the values here, and they represent the customer's last name, this here. And the customer's yearly subscription to the train service. For now, these subscriptions, we'll just assume that they are an instance of a case class that has no field. So don't worry about exactly how they're implemented. While on the other hand, the locations dataset is a little simpler. It's just one Pair RDD with a string as a value. In this case, the key is the customer number. And the value is the name of the city that the customer most frequently travels to. So in both Pair RDDs, the keys are the customer numbers. Also note that locations is bigger than abos. There are more elements in locations. So both of these datasets have different sizes.

Example Dataset... (3)

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

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



And again, you can imagine that the abos dataset here comes from the CFF's customer database of subscriptions. And the location dataset you could imagine could come from months of collecting data about how registered users are using the train service from the mobile application. So on the smart phone app for buying train tickets. Importantly, this means that it's possible to be a registered user of the smartphone app. So you can be in this data set here. But you don't necessarily have to have a yearly subscription like the 50% discount card called the DemiTarif. So it means it's possible to have people in this dataset that don't yet exist or don't exist at all in this dataset because they don't have any kind of yearly subscription. And vice versa, it's possible for people in the abos dataset here to not be in the locations dataset because they don't have or use a smartphone, for example, to buy their train tickets. Maybe they just buy their train tickets using a regular paper ticket machine. So that's also possible. So this is something like a real dataset because there are little imperfections between the two datasets that we have to somehow take care of if we'd like to merge them together somehow.

Inner Joins (join)

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

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

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

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

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

val trackedCustomers = ???
```

We'll be able to handle these cases using one of a few different choices of join operations in Spark. The first one we'll look at is called inner join. We'll be able to handle these cases using one of a few different choices of join operations in Spark. The first one we'll look at is called inner join. The simplest explanation of what this operation does is as follows. Inner join returns a new RDD containing combined pairs whose keys are present in both input RDDs. What does this mean? The signature here can helps out a little bit. So this method join here can be called on a Pair RDD passing another Pair RDD as a parameter. It assumes that both RDDs have keys of the same type, as we can see here. And that they can have values of different types. In this case, values of type V and W. But what this operation does is it returns a new Pair RDD where the value of the resulting Pair RDDs is itself another pair containing both values of the two input Pair RDDs. What's most important to remember is that the inner joins are special because the resulting pair RDD will only contain key value pairs per keys are present in both RDDs. That means if this join is kind of lossy, if there's a key in one RDD that's not in the other, it gets dropped. Of course, these things are always much clearer with an example. So let's use the dataset that we just looked at. How do we create new RDD which can tell us which customers one, have both a subscription? And two, will also use the CFF smartphone app so we can keep track of which cities they regularly travel to? So how would you implement this value called trackedCustomers here? What would you do to do this? I'll give you a second to think about it.

Inner Joins (join)

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

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

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

So you can probably guess that we're going to use the join method that we just introduced, right? This one called join. But first question is which pair RDD should we call this join on? Do we call it on locations, and then do we pass abos as an argument? Or do we call it on abos and pass locations as an argument? Does it matter?

Inner Joins (join)

Example continued with concrete data:

abos

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

locations

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

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

The answer is that no, it doesn't matter, we can just do abos.join(locations). And we can look at this little visualization here to get an idea of what's happening to the actual data when we do this join.

Inner Joins (join)

Example continued with concrete data:

abos

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

locations

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

We want to combine both RDDs into one:

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

So remember, the goal is to combine both Pair RDDs into one Pair RDD. And in particular, what we want to do is we want to put customers who have both subscription info and location info into the resulting Pair RDD.

Inner Joins (join)

Example continued with concrete data:

abos

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

locations

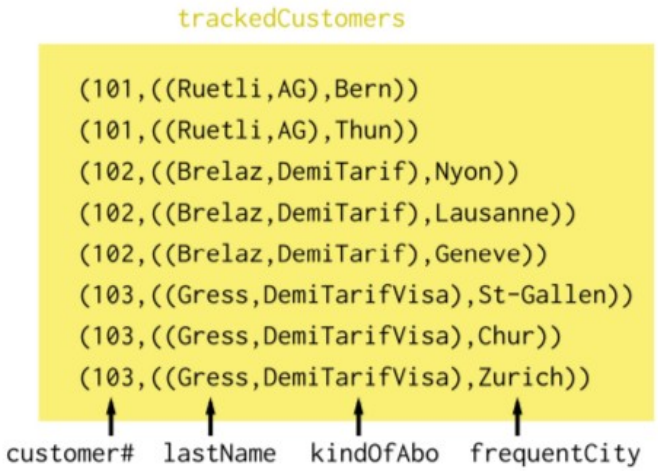
```
(101, "Bern"),
(101, "Thun"),
(102, "Lausanne"),
(102, "Geneve"),
(102, "Nyon"),
(103, "Zurich"),
(103, "St-Gallen"),
(103, "Chur")
```

We want to make a new RDD with only these!

Visually, that means that we want to make an RDD with only these highlighted elements inside of it because they exist in both RDDs. So 101, 102, 103, these customers, we've got them also in the locations RDD here. So customer 101, customer 102, customer 103 are here. So we want to make a new RDD with only these elements in it.

Inner Joins (join)

Example continued with concrete data:




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

And this is what our resulting Pair RDD looks like after calling abos.join(locations). Remember that the value of the new key value pair is another pair representing both input values. So in this case, the first element of the value pair, so here is the values. The first element is another pair representing the last name of the customer and their type of subscription. And the second element of the value pair is the city that they frequently travel to. Do you notice anything else weird about these results? They're a kind of sort of duplicates, right? Before we only had one element for this customer (101). But now we have one for each element that was in the locations RDD. This is because, if we go back and look at the original two datasets, some customers like 102 and 103, they have a handful of cities that they like to frequently visit. But in the abos dataset, we have just one entry per customer number. So the resulting Pair RDD, that customer info is duplicated for each different frequently-visited city by that customer. So this is how the data is merged together.

Inner Joins (join)

Example continued with concrete data:

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

What happened to customer 104?

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

And remember joins are transformations. Just like in all of the other sessions so far, if we actually want to see the resulting dataset like we saw in the previous slide, we actually have to invoke some kind of action to kick off the computation. And again, like we've done in previous sessions, we're just going to call collect on the RDDs here, on this trackedCustomers RDD. It's a pair RDD, and then once we get back this collection, in this case an array, we'll do foreach on it. And println to just see what the results look like. But wait a minute, what happened to customer 104? Look back at this abos dataset on one of the previous slides. Notice that this person is gone. Remember, there was this customer here. Customer 104 doesn't exist in the output. See, this person is missing. Customer 104 does not occur in the result because there's no location data for this person in the location dataset. Remember that inner joins require keys that occur in both source RDDs. Else they're dropped from the result.

Outer Joins (leftOuterJoin, rightOuterJoin)

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

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

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

(Notice the insertion and position of the `Option`!)

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

Which join do we use?

Let's now shift gears a little bit and visit the other kind of join, outer joins. Said most simply, outer joins return a new Pair RDD containing combined pairs whose keys don't have to be present in both input Pair RDDs. So that means this kind of join is particularly useful for figuring out how to deal with missing keys between Pair RDDs. This is why there are two kinds of outer joins, left outer joins and right outer joins. Because it lets you decide which RDDs keys are the more important to keep, the ones on the left or the ones on the right of the joint expression. To better understand, let's have a look at the type signatures here. They look a lot like the type signature of the inner join with the exception of the Option type here on the return type. This means that if a key isn't present in both input RDDs, the optional value could simply be none. So you have an entry still, but instead of being some and then the value (看了後面例子就知道是甚麼意思了), you have then just none. And notice for the leftOuterJoin here, so on this one, the option is on the second element of the value pair in the result. And on the rightOuterJoin, the option is on the first element of the value pair in the result. This is how we decide which input RDD we prioritize for the keys. As usual, it's always nice to try to better understand these operations with a concrete example. So let's go back to our CFF dataset. And let's try to solve a slightly different problem. So let's say the CFF wants to know for which subscribers it has collected location information. For example, we know it's possible that somebody has a subscription like a DemiTarif. But doesn't use the mobile app and will always pay for tickets with cash at a machine. Which of these two outer joins do we use?

Outer Joins (leftOuterJoin, rightOuterJoin)

Example continued with concrete data:

abos

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

locations

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

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

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

Again, let's return to our visual depiction of the data. So we want to combine both these Pair RDDs into one Pair RDD. And we want to know for which subscriber, that means for which people in this abos dataset, that we've also collected information for about location. So which kind of outer join do we choose to compute this abos with optional locations value?

Outer Joins (leftOuterJoin, rightOuterJoin)

Example continued with concrete data:

abos

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

locations

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

We want to make a new RDD with these!

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

We can also highlight the elements that we want to keep when doing this join. So we can highlight the elements that we want to keep when doing this join. We want to make a new RDD with these elements that are in yellow. So which join do I choose? And on which Pair RDD do I poll that join operation? Does it matter? Think about it and give it a try yourself. Go back and look at the signatures of the outer joins if you need to. And when you look at those signatures, ask yourself, which part of the resulting value pair should be the optional part? Well, the answer is that the elements from the location dataset should be the optional part of the resulting value pair. Because, remember, we care about subscribers, so we prioritize the subscriptions RDD called abos. So with this information, we can now choose which join you want to use and on which RDD we want to call that join on. So since the important RDD is the subscriptions RDD called abos, that means we call join on abos. Now we just have to figure out which outer join method to use. So let's look at the type signatures. The leftOuterJoin method makes it possible for the second element of the resulting pair to be optional.

Outer Joins (leftOuterJoin, rightOuterJoin)

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

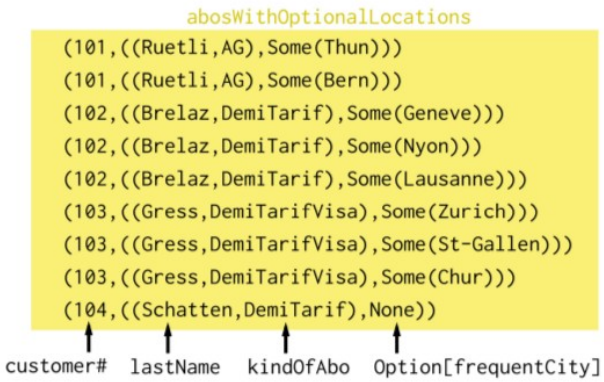
Which join do we use?

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

So the answer is that we call leftOuterJoin on abos and we pass locations in as an argument.

Outer Joins (leftOuterJoin, rightOuterJoin)

Example continued with concrete data:




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

Here are the results visualized. Like last time, the key is the customer number and the value is a pair. Where the first element is the subscription data here, last name and subscription type. And the second element is the optional value for the customer's frequently-visited cities, so here.

Outer Joins (leftOuterJoin, rightOuterJoin)

Example continued with concrete data:

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

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

And of course, this is what it looks like when we use collect and foreach to actually kick off that computation. Because we can't forget that the leftOuterJoin method, like all the other join transformations, they're transformations. So we have to do something like collect, some kind of action to start the computation. So said another way, since we used a leftOuterJoin, keys are guaranteed to be kept in the left input Pair RDD. This is why we can see customer 104 here. That's because this customer has a subscription. He or she has a DemiTarif here which is on the left side of the join. So the element must be kept even though there is no location data for this key for this customer in the location dataset. Hence, why we see the value None here for the most frequently-visited cities.

Outer Joins (leftOuterJoin, rightOuterJoin)

We can do the converse using a rightOuterJoin.

abos

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

locations

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

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

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

Let's flip this problem the other way around. Let's imagine instead that the CFF wants to know for which of its smartphone app users it has subscriptions for. You can imagine that perhaps the CFF would like to offer a discount maybe on a subscription to one of these users. So maybe the CFF would like to identify the people who use the mobile app, but don't yet have a DemiTarif subscription. And they maybe want one. So which OuterJoin should be used and which RDD should we have on the left and on the right side of the join operation to find these users?

Outer Joins (leftOuterJoin, rightOuterJoin)

We can do the converse using a rightOuterJoin.

abos

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

locations

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

We want to make a new RDD with only these!

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

Visually, that means we want this part of the dataset. That is, we don't care about customer number 104 because that customer doesn't use the smartphone app. Therefore, we want to make a new RDD with only these elements. Which join should we use, and on which pair RDD should we call the join on?

Outer Joins (leftOuterJoin, rightOuterJoin)

We can do the converse using a rightOuterJoin.

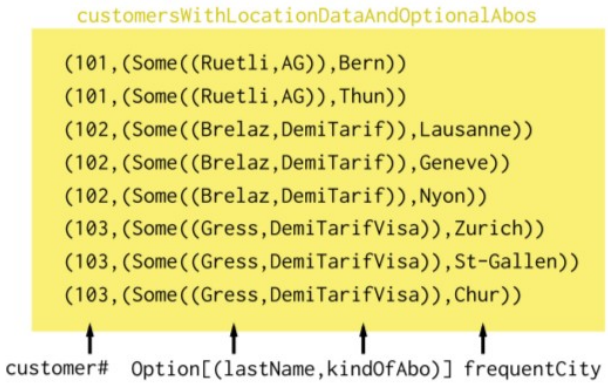
Example: Let's assume in this case, the CFF wants to know for which customers (smartphone app users) it has subscriptions for. E.g., it's possible that someone uses the mobile app, but has no demi-tarif.

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

Of course, the slide gives the suggestion away. Let's just try to use the rightOuterJoin. Why? Remember, the signature of the two outer joins. For the rightOuterJoin, the optional element is the first element of the resulting pair. In our case, that would be the value from the abos dataset, which is the less-important dataset in the join. Since we're focusing on smartphone users, we want to make sure that we keep every key/element of the locations datasets. Therefore, we should do abos.rightOuterJoin(locations).

Outer Joins (leftOuterJoin, rightOuterJoin)

Example continued with concrete data:




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

This is what the resulting combined dataset looks like after calling abos.rightOuterJoin(locations). This result isn't the most interesting result because there is no customer which uses the mobile app, but which doesn't have a subscription already. If there was, we would see at least one element, or rather

than this optional Some here, Some, last name, subscription, we would just see type None. And then the cities that that customer frequently travels to.

Outer Joins (leftOuterJoin, rightOuterJoin)

Example continued with concrete data:

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

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

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

Here it's what it looks like printed out after calling collect for each print line. What's important to note is that we lose customer number one 104 again, you see? Because that customer number is not in the right side of the join, so it gets dropped. We don't need it, so it doesn't end up in the resulting join dataset. Phew, so those were the different kinds of joins in Spark.