Shuffling: What it is and why it's important

本黑體字之管理下的所有內容的意思是: 若能用 reduceBy 做到跟 groupBy 相同的事, 就盡量用 reduceBy, 不用 groupBy.

Shuffling: What it is and why it's important

Big Data Analysis with Scala and Spark

Heather Miller

In this session, we're going to try and wrap our minds on something that is inevitable that we might not considered when writing Spark programs thus far, it's called shuffling and it can happen quite a bit. You never call a method called shuffle but as we're going to see there are many methods which transparently cause shuffle to happen. We're going to dive in to some of those scenarios in this session.

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

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

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

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

Shuffles Happen

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

So, let's first start by looking at a groupBy or groupByKey operation. For a moment just try to think about what these methods do? And what might happen when one of these methods are called? Importantly, you can't forget that our data is distributed. Now, think again about what a groupBy or groupByKey does now with the fact that our data is distributed in your mind. Well, what about when you've been using Spark on your own? Did you notice anything when playing around with Spark for your programming assignments, about the types when you used groupBy, or groupByKey? Here's a very quick example. Here we just make a pair RDD, and then we call groupByKey on it. And this is the return type that Spark gives us, it's something called a ShuffledRDD. Hmm, that's weird. To do a distributed groupByKey, we typically have to move data between nodes so the data can be collected together with its key in a regular, normal single machine Scala collection. Remember that groupByKey collects all of the values associated with the given key and stores them in that single collection. That means that data has moved around the network and doing this, moving the data on the network is called shuffling. What's important to know is that shuffles happen. They happens transparently as a part of operations like groupByKey. And what every Spark program are learns pretty quickly is that shuffles can be an enormous hit to performance because it means that Spark has to move a lot of its data around the network and remember how important latency is.

Grouping and Reducing, Example

Let's start with an example. Given:

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

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

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

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

Let's try to better understand what's really happening when we do operations like groupByKey. As usual, we'll look at an example to help clarify. Let's assume we have a case class called CFFPurchase which contains three fields. And integer for the costumerId, a String for the destination city, and a Double representing the price of the ticket that's been purchased as a part of this CFFPurchase and remember the CFF is the Swiss train company. Assuming that we have an RDD that's full of purchases made in the past month, let's try to calculate how many trips and how much money was spent by each individual customer over that last one month period.

Grouping and Reducing, Example

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

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

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

So with all the knowledge that you have so far what would you do? Which methods would you use?

We've been working on pair RDDs quite a bit now, so perhaps the first step is to make a pair RDD. Okay, but what other methods do you use? I'll give you a moment to try and figure it out yourself. Since the end goal of this example is to try and figure out how many trips each customer has made this month and how much money they've spent over the month, the destinations that they've gone to in each purchase isn't important, so we'll drop that when we make our new pair RDD here. We'll have as the key, the customerId and as the value the price of the purchase that this case class instance represents. Next, since the goal is to focus on each individual customer, and to then calculate the number of trips they've made, and how much money they've spent in the past month, it would make sense to group together all of the purchases per customer. So in that case, we reach for good old groupByKey which returns a Pair RDD with a collection of values that correspond to the values that go along with each key. So now we have a Pair RDD where customerId is the key and then a collection of all of the purchase prices of that customer's purchases over the past month is the value. So we have almost everything we need now. The last step is just to figure out how many purchases were made per customer and how much money was spent in total by each customer. What method should we use next? We can use a simple map, or we can even use mapped values. Here, I've used map but mapped values also works just fine. All we have to do is make a new pair containing the number of purchases as the first element of the pair which is easy to do, we can just call size on the collection of values. And for the second element of the pair, we just need to sum up all of the prices in the collection so we can simply call the method sum on the collection of values. And finally, since none of the methods we called so far are actions yet, we have to call an action just to kick off the computation. So in this case, we call collect, to collect all of the result onto the master node.

Grouping and Reducing, Example – What's Happening?

Let's start with an example dataset:

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

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

Now that we understand an example solution to this problem, let's step through what's really happening with the data given some sort of example data set. So here, let's imagine our purchases are this short lists of instances of CFFPurchase. How might the cluster look with this data distributed over it?

Grouping and Reducing, Example – What's Happening?

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

Starting with purchasesRdd:

So let's start with the data in the purchasesRdd. Let's imagine that we have three nodes, this one, this one, and this one. And we split up our data over these three nodes evenly. So since we have six data elements in the purchasesRdd, that means, for the sake of this example, we can put two elements on each node. So two here, two here, and two here.

Grouping and Reducing, Example – What's Happening?

What might this look like on the cluster?

Let's for a second imagine that the map function (即前面的 map(p => (p.customerId, p.price))) is executed on this data that's producing a set of pairs on each node. Because remember, after we had these CFFPurchases, what we did was we created a pair RDD. So assuming that was evenly executed, it would look like this. So remember in our code example, we got rid of the destination cities and we kept only the customerId number and the price paid per purchase as the key in the value in our key value pairs. So now we went from six instances of CFFPurchase to six instances of key value pairs, two on each of the three nodes.

Grouping and Reducing, Example

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

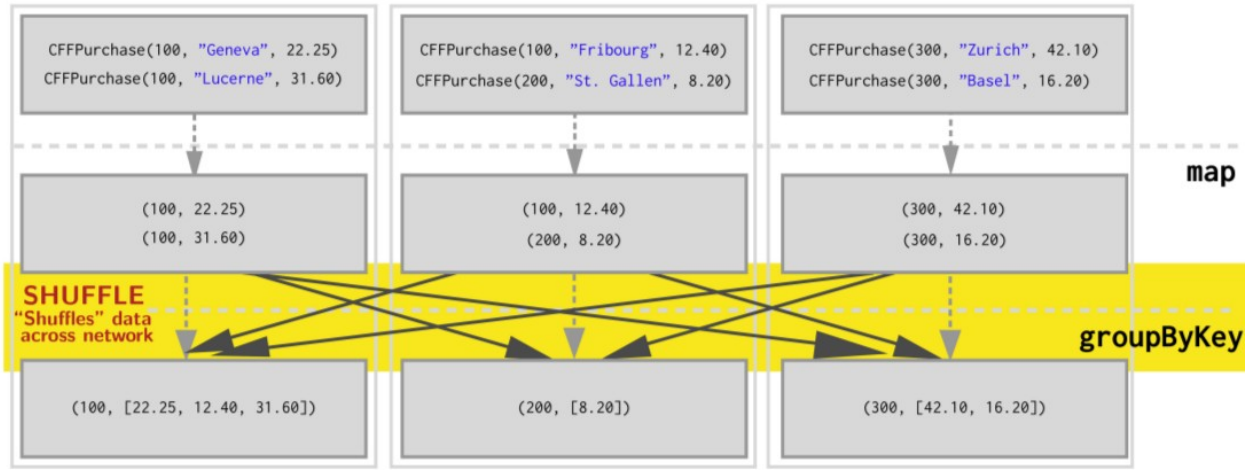
```
val purchasesRdd: RDD[CFFPurchase] = sc.textFile(...)


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

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

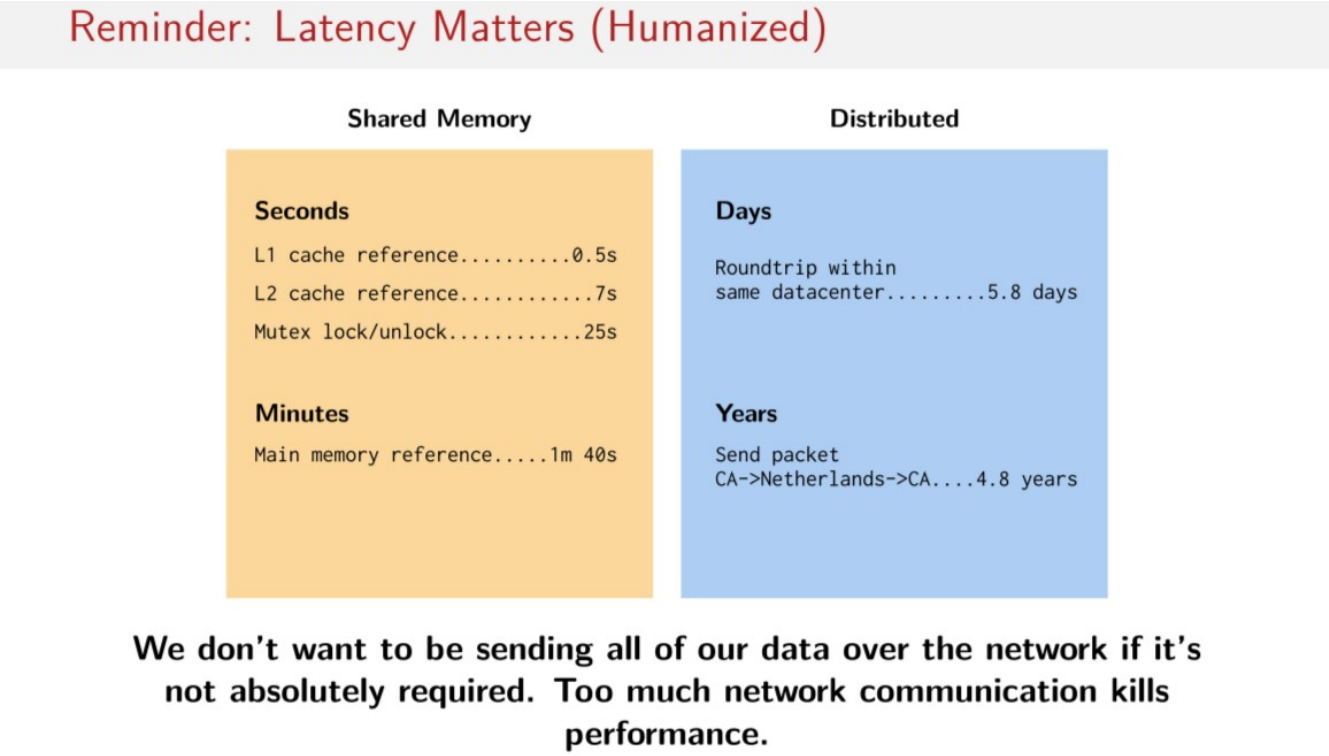
Now if we look back at the code in the example, the next step was to do a groupByKey. And remember, groupByKey results in a single key value pair per key, so think about it. Therefore, a single key value pair cannot span across multiple worker nodes.

Grouping and Reducing, Example – What's Happening?

What might this look like on the cluster?



So let's go back to our visualization of the cluster. What does groupByKey look like in this case? Remember, we have concrete instances of data, so how would this data have to move around? Well, as you might have guessed, to create collections of values to go with each unique key, we have to move key value pairs across the network. We have to collect all of the values for each key on the node that the key is hosted on. So in this example, we've assumed that since there are three unique keys and three nodes, each node will be home to one single key. So we put 100 on this node, 200 on this node, and 300 on that node. So then we move around all of the key value pairs so that all purchases by customer number 100 on this node and all purchases by customer number 200 are on this node and all purchases by customer number 300 are on this node and they're all in this value, which is a collection. This highlighted part here is where all of the data moves around on a network. This part of the operation is the shuffle.



Now I'm just going to step back to one of the slides from the beginning of the course about latency. Remember the humanized differences between operations done in memory and operations that require sending data over the network? The humanized difference between these two kinds of latency was on the one hand seconds to minutes and on the other hand up to years. With these numbers in mind,of course we don't want to be sending all of our data over the network if it's not absolutely positively required. Having to do a lot of network communication, kills performance especially if it's being done unnecessarily. So this is why shuffling is bad. We want to do as little shuffling as possible because of these orders of magnitude.

Can we do a better job?

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

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

上圖中的 reduce 不是 MapReduce 那樣自動弄的, 而是要我們自己調用 reduceByKey 函數.

So how can we do a better job? What can we do differently to make this example more performant? After all, if you want to do a groupByKey kind of operation, we definitely have to move data over the network. But can we somehow do it in a more efficient way? Perhaps we can reduce before doing the shuffle. This could potentially reduce the amount of data that we actually have to send over the network.

Grouping and Reducing, Example – Optimized

We can use `reduceByKey`.

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

Signature:

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

To do that, we can use the reduceByKey operator, do you remember this one? We learned about it in

one of the previous sessions. And remember, you can think of it as a combination of first doing groupByKey and then reducing overall the values that were grouped by that key in those collections. And I also told you, back when we cover this method, that this one was much more efficient but I didn't show you how yet. We'll look at exactly how this is more efficient to the next example. But first, let's recall reduceByKey's type signature. Remember, since it operates on the values that you assume are already grouped by some key, we focus only on the types of that value. So the function passed to the reduceByKey operator, only operates on the key value of the value pair.

Grouping and Reducing, Example – Optimized

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

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

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

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

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

So let's go back to our earlier example. Remember, our goal was to figure out how many trips and how much money was spent by each individual customer over the course of the month. So if we were to replace our call to groupByKey with the call to reduceByKey instead, what will we have to pass to reduceByKey? What would the function be here that we'll have to calculate the total number of trips and the total amount of money spent by each customer? Note, to make things a little easier for you, I changed the function that I passed to map here. So it's a little different than it was in the previous example. Now, instead of a Pair RDD with only the customerId and price, I've also added an integer 1 here to make it easier to do the count of the purchases later. So given that, what should the functional literal be that we pass to reduceByKey in order to get a result that looks like this? A pair of customerId with a value that itself is a pair, of trips, total money spent.

Grouping and Reducing, Example – Optimized

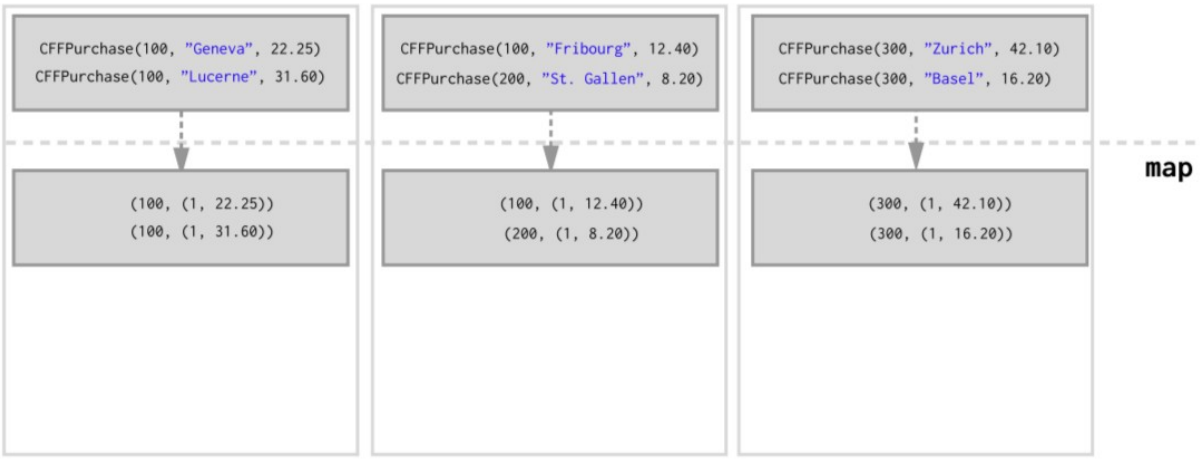
```
val purchasesPerMonth =
  purchasesRdd.map(p => (p.customerId, (1, p.price))) // Pair RDD
              .reduceByKey((v1, v2) => (v1._1 + v2._1, v1._2 + v2._2))
              .collect()
```

1+1 price + price

Remember that types of I just showed you for the reduceByKey operator. We're reducing over the values per key. And since our values are types integer, Double, Where here we have the price and here we have an integer that we can count up, the function that we pass to reduce by key must reduce over two such pairs. I'll give you a moment to try and come up with the solution with this information now available. The solution is pretty simple. Since the values that we're reducing over are pairs, where each element of the pair must be summed with the same elements of the adjacent pairs, all you have to do is sum up the two first elements of the JSON pairs, remove the sum up the two second elements of the JSON pairs, and finally, kick off the computation with .collect, here. So all we're doing is summing up the first two elements and summing up the second two elements in the pairs. So we're adding together 1 + 1, in this case, and some price + some price, in this case. And after that, you just have to invoke some kind of action to kick off computation and that's it. So what might that look like on the cluster that we just visualized?

Grouping and Reducing, Example – Optimized

What might this look like on the cluster?



Going back to our visualization, note that I updated the result of the map. So, we now have what we've been working with here with this extra integer there. What happens with this data on the cluster for this code example now?

Grouping and Reducing, Example – Optimized

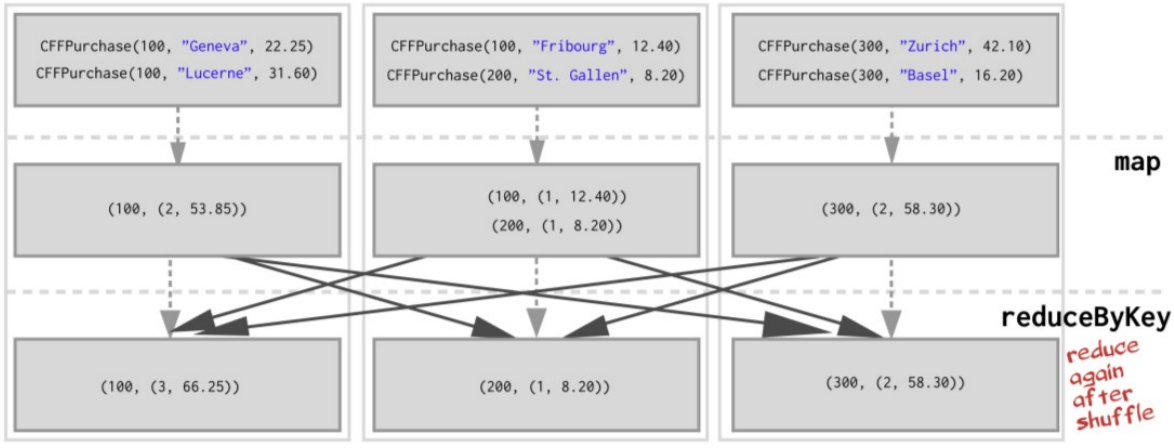
What might this look like on the cluster?

注意看上圖跟上上圖相比 之變化

Well, since reduceByKey reduces the data on the mapper side first, the pairs on our three nodes are reduced and they look like this. So here's before, here's after, so what we had was two elements with key 100 on this node, two elements with key 300 on this node. We could reduce them together into a single element that looks like this. Notice that the logic in our reduceByKey was essentially applied to the mapper side first. Imagine if this was a real data set with millions or billions of elements in each node, now we have at most one key value paired per node. So that's potentially a very large reduction in the amount of data that maybe we have to shuffle.

Grouping and Reducing, Example – Optimized

What might this look like on the cluster?



The idea is that hopefully we're shuffling less data now and then we do another reduce again after the shuffle. And in the end, we should have the same answer, but we should have arrived at that answer in considerably less time because the goal here is to reduce the amount of network traffic caused when we do one of these operations.

Grouping and Reducing, Example – Optimized

What are the benefits of this approach?

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

This can result in non-trival gains in performance!

Let's benchmark on a real cluster.

So just to summarize, what are the benefits of this approach? Well, by reducing the dataset first, we can reduce the amount of data that's sent over the network during the shuffle which could mean pretty significant gains in performance. And to make this a little more concrete for you I performed this computation, the exact same code on a real cluster and we can easily see the gains in performance.

groupByKey and reduceByKey Running Times

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

purchasesPerMonthSlowLarge: Long = 100000
Command took 15.48s

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

purchasesPerMonthFastLarge: Long = 100000
Command took 4.65s
```

This is done with billions of generated key value pairs representing the same data sets and the same code we just saw on the previous slides on a cluster of I believe six nodes. This was run in a Databricks notebook and what we can see here is the amount of time it took to do the same computation with a groupByKey versus with a reduceByKey and as we can see, the groupByKey variant is exactly what we saw on one of the previous slides. The groupByKey variant takes 15.48 seconds to complete. While the variant using reduceByKey like I said, same as we had on one of the previous slides takes only 4.65 seconds to complete. In this example, the reduceByKey version of this program is up to three times faster than the groupByKey version. Imagine if this was an hours long computation, this would mean a lot of save time on our big data set.

Shuffling

Recall our example using groupByKey:

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

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

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

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

So I think we have a little more intuition now about how all of this works. The bottom line is that if we use operations like groupByKey a lot, the active grouping all of the keys with their values requires us to move the values to be on the same machine with their responding keys. The one big question that we haven't yet asked is, how on Earth Spark knows which key to put on which machine? We just assume that Spark is doing something sane and is spreading the data out evenly but, how does that really work? Well, the answer is partitioning. And how this works we'll cover in depth int he next few sessions. And partitioning can make a big difference to how fast your job completes.

Partitioning

Partitioning

Big Data Analysis with Scala and Spark

Heather Miller

In this session we'll cover something called partitioning which comes in to play when shuffling data around your cluster. Partitioning your data intelligently can often give you a lot of time when running computations. It's important to understand in general with distributed systems and in particular with dealing Spark RDDs.

"Partitioning"?

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

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

We concluded the last session asking ourselves,

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

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

In the last session we focused most of the session on the specific example involving the groupByKey operation. We saw that using this operation causes data to be shuffled over the network. This is because to group all of the values with their corresponding key, we've got to move all of the values to the same machine. But we ended last session asking ourselves, well wait I understand we have to move data around the network. So some data is all on the same machine, but how do we know which machine to put which key on? So before we try to optimize that example that we saw in the last session any further. Let's take a detour to learn more about what partitioning is, and how it works in Spark.

Partitions

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

Properties of partitions:

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

Two kinds of partitioning available in Spark:

- Hash partitioning
- Range partitioning

Customizing a partitioning is only possible on Pair RDDs.

Let's first start with partitions, what are they? Simply put, the data within an RDD is split into many partitions, and partitions are very rigid things. Most importantly, they never span multiple machines, this is super important. Data in the same partition is always on the same machine. Another point is that each machine in the cluster contains at least one partition. Sometimes more, sometimes exactly one. Actually, the number of partitions is configurable, by the way. We'll learn about when and how the number of partitions can be changed by the user. Though by default, when a job starts the number of partitions is equal to the total number of cores on all executor nodes. That's the default. So for example if all of the machines in your cluster have four cores and you have six worker nodes then that means the default number of partitions you can start with may be 24. And importantly spark comes with two out of the box kinds of partition which makes sense for different sorts of applications. These are hash partitioning and range partitioning, we'll talk about these in more detail in the next few slides. However, it's important to note that all of this talk about customizing partitioning is only possible when working with Pair RDDs. Since, as you'll see, partitioning is done based on keys, so we partition based on keys, hence, we need a Pair RDD to do that.

Hash partitioning

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

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

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

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

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

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

So let's start with hash partitioning. To illustrate this first kind of partitioning let's return to our groupByKey example that we saw in the previous session. As we saw on the last session by defaulting this example hash partitioning is used but what is it? How does it work? What does it look like? The way it works is like this. Since groupByKey knows it has to move all of the data around the cluster, its goal is to do that as fairly as possible. So the first thing it does is it computes the partition p for every tuple in the pair RDD. So we start by getting the key's hash code And then we modulo that with the default number of partitions. So we said it was 24 in the last slide. And whatever the answer is of this computation, is the partition that that key goes onto. Then when we actually do the hash partitioning, the tuples in the same partition are sent to the machine hosting that partition. So again the key intuition here is that hash partitioning tries to spread around the data as evenly as possible over all of the partitions based on the keys.

Range partitioning

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

- Examples: `Int`, `Char`, `String`, …

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

Using a range partitioner, keys are partitioned according to:

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

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

The other kind of partitioning is called range partitioning. This is important when some kind of order is defined on the key so examples would include integers or charge or strings for example and if we think about our previous examples where. We were working with pair RDDs that had keys that were integers, we could hypothetically range partition these keys. For these kinds of RDDs with keys that can have an ordering, range partitioning could be an efficient choice for partitioning your data. Intuitively though, that means when you're using a range partitioner, keys are partitioned according to two things. They're partitioned according to some kind of order, so like I said a numerical order electrographical ordering as well as a set of sort of ranges for the keys. So then you have groups of possible ranges that the keys can fall within. And that means that tuples with keys in the same range will appear on the same nodes.

Hash Partitioning: Example

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

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

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

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

$p = k \% 4$

The result is a very unbalanced distribution which hurts performance.

Let's look at a more concrete example each kind of partitioning must start with hash partitioning, so consider we have a Pair RDD with these keys. So we don't care about the values for now but these the keys that we have in our pair RDD. Let's say our desired number of partitions is 4. Also just creating simple let's assume that the hashCode function is just the identity function for this example. And let's remember that you compute the partition what we do is we call the hash code on the key. Which like I said, it's just the identity function and we modulo that with a number of partitions. So in this case this will simplify to just the key, modulo 4 for the sake of this example, so let's compute the partitions that these keys belong in. So if we try to compute the partitions for each of these keys and we use the formula on the slide previous like this, And something wonky happens, this hash partitioning distributes our keys as follows amongst our four partitions. Remember the goal of hash partitioning is to try and evenly spread out the keys, though it's possible to have situations like these where you have an unlucky group of keys that can cause your data. When you're clustered to be unevenly skewed, and for some nodes to have a lot more data, and for some nodes to have none, or a lot less. Of course this means potentially bad performance, because a job could be evenly spread out on four nodes. And in this case, it's basically just spread out on one node, right now, so it's not really very parallel.

Range Partitioning: Example

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

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

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

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

The resulting partitioning is much more balanced.

In this case, since we know the hash partitioning is actually skewed, and since our keys have an ordering and are non-negative. We can use range partitioning to improve the partitioning, and to even it out significantly. So assuming that there are a set of ranges for four nodes, for example, for customer IDs 1 to 200 on the first node, customer IDs 201 to 400 on the second node. Customer IDs 401 to 600 on the third node, customer ID 601 to 800 on the fourth node. This would enable us to distribute the keys more evenly across our partitions. So here's a scenario where we can more easily distribute our keys through out the cluster, that is we just put the keys in their corresponding ranges. So all keys, between 1 and 200 on the first node, all keys between 201 and 400 on the second node, etc. This is clearly much more balanced and of course we can imagine it being much more performant than before because the work is more evenly spread out.

Partitioning Data

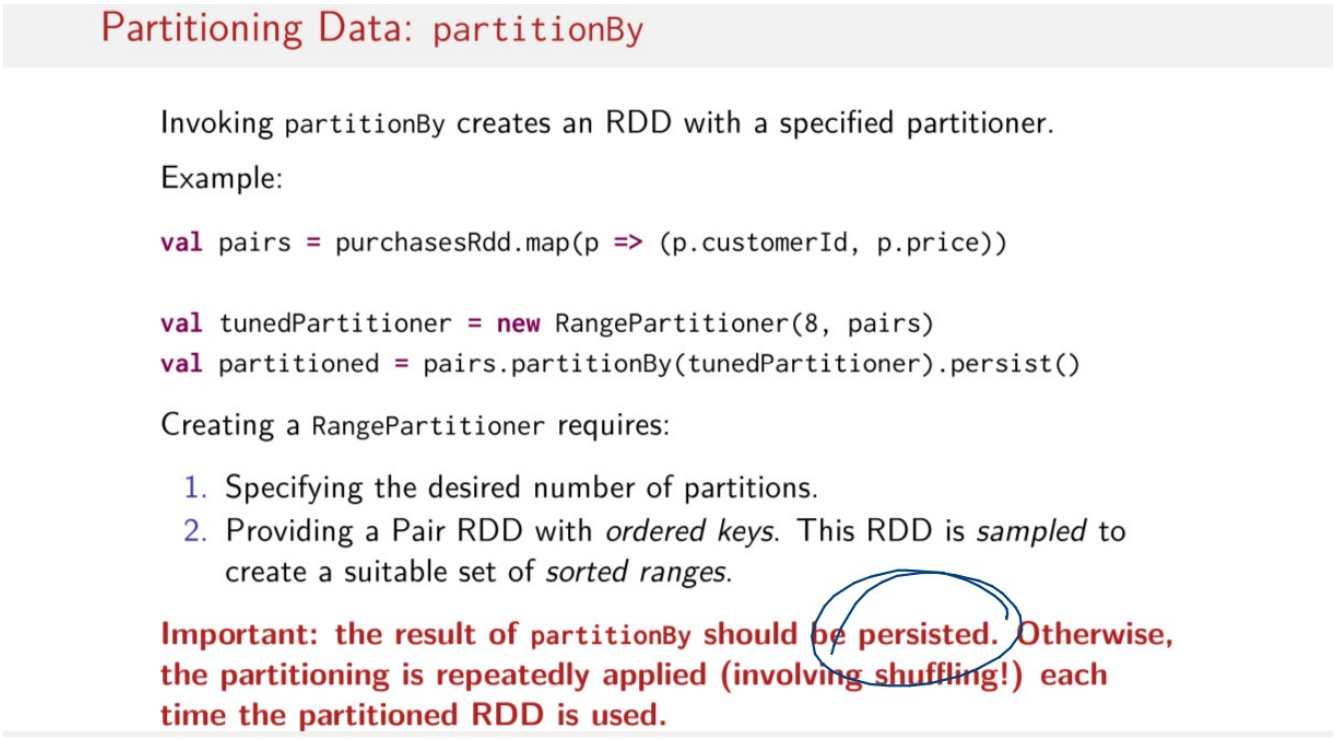
How do we set a partitioning for our data?

There are two ways to create RDDs with specific partitionings:

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

Okay but what if I wanted to customize how my data is partitioned? We illustrated the concepts behind the hash and range partitioning but we haven't seen yet how to customized and set partitioners for any of our spark jobs yet. Well there are two ways to create RDDs with specific partitioning's. First you can exclusively, called the method partitioned by on an RDD and you can pass in an argument which is an instance of some kind of partitioner. Or second, you could just keep track of the transformation that you use in your RDDs because certain transformation use certain kinds of partitioners. So if you know the partitioner associated with the transformation you're using, you can also customize the partitioning of

your data that way.



Let's first look at the partitionBy method. So if you try to customize your data partitioning by using the partitionBy method, the result is that partitionBy creates a new RDD with a specified partitioner. So let's imagine for a second that we have this RDD of purchases that we've been working with in the previous sessions. And let's say that we want to use a rangePartitioner on it now to do this, we first need to create an instance of a rangePartitioner here. This is done by calling new rangePartitioner and passing in the number of partitions that we want. In this case 8, and we'll also have to pass reference to an RDD that we'd like to have partitioned. In order to come up with these ranges, we need to sample the RDD that we'd like to have partitioned, so that's why we have to pass in pairs here. Sparkle take pairs and sand ballot to find out what the best ranges are. Now we simply invoke partitionBy on the pair's RDD, and we pass in the new ranged partitioner that we just created. Do you see anything else interesting on this slide? I'll give you a moment to think about it. Yes, it's this call to persist here. Why would it be desirable to call persist here? Well, the answer is new semantics and tendency to reevaluate chains of transformations again and again and again. This data would be shuffled over the network and partitioned again and again and again if we don't tell Spark. Basically what we're doing here is we're saying hey, once you move the data around in the network and repartition it just keep it where it is, persist it in memory. Else you'd easily find yourself accidentally re-partitioning your data in every iteration of a machine running algorithm, completely unknowingly. So just to repeat, to create a range partitioner you must simply specify the desired number of partitions, and then you must provide a pair RDD, with keys that are ordered. So that Spark is able to sample the RDD and create a suitable set of ranges to use for partitioning based on the actual data set. And again, I can't remind you enough, the results of partitionBy should always be persisted. Otherwise, partitioning is repeatedly applied, which involves shuffling each time the RDD is used and of course you don't want this to happen.

Partitioning Data Using Transformations

Partitioner from parent RDD:

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

Automatically-set partitioners:

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

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

Earlier in the session, I mentioned that it was possible to partition using transformations as well. There are two ways partitioners can be passed around with transformation. In the first case, and probably the most common case, partitioners tend to be passed around via parent RDDs. That is paired RDDs, which are the result of a transformation on an already partitioned paired RDD, is typically configured to use the same HashPartitioner that was used to construct its parent. The other case where transformations can actually provide a new partitioner is when some transformations automatically result in an RDD with a different partitioner. Of course this is usually when it contextually makes sense so for example if you use the sortByKey operation, a RangePartitioner is used because you need ordered keys in order to sort them. They make sense that the keys once sorted are partitioned such that the similar keys are in the same partition. Conversely, the default partition when using groupByKey hash partition like we saw in earlier example.

Partitioning Data Using Transformations

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

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

All other operations will produce a result without a partitioner.

Here are list of transformations that either hold on to or propagates them kind of partition. This is

important to remember because it could bite you for performance reasons later. These are operations that could result in a completely different partitioning for your data. So it's important to remember that by just using some kind of transformation, that something like this can happen transparently under the hood. One big thing you should notice though is that a certain function that you use all the time is not on this slide. That's right map is missing and so is flatMap. Why aren't on this list? Interestingly though, flatMapValues and mapValues are on this slide and filter as well. And in these cases, these three operations hold on to their partitioner if their parent has some kind of partitioner defined. Otherwise any other transformation operation that's not on this slide will produce a result that doesn't have a partitioner, and that includes map and flatMap. If you use map or flatMap on a partitioned RDD that RDD will lose its partitioner. That could really screw things up just think about it for a second. If you've gone to great lengths to try and organize your data in a certain way across your cluster and then you use an operation like flat map. Then you lose all of that partitioning and your data might be all moved around again but let's focus on this point right here for a second.

Partitioning Data Using Transformations

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

Why?

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

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

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

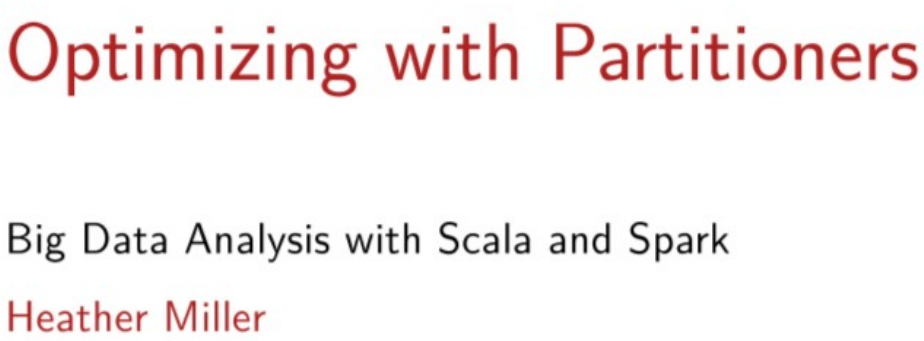
In this case, if the `map` transformation preserved the partitioner in the result RDD, it no longer make sense, as now the keys are all different.

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

Why is it that all other operations will produce a result that doesn't have a partition? Or why does that make sense? Think about the map transformation for a moment. Assuming we have a hash partitioned RDD that we intend to map on, why would it make sense for the resulting RDD to lose its partitioner in its result RDD? Think about all of the things you could possibly do in a map operation and remember that partitioning only makes sense for Pair RDDs which are key value pairs. I'll give you a moment to think about it. Well, that's because it's possible for map, or flat map, to change the keys in a Pair RDD. Here's an example. We can take a very intelligently crafted data set, called RDD in this case, and we can accidentally replace all of the keys in the entire data set with the same key. Which is just the story doh. We could do this intentionally or accidentally doesn't matter. So therefore if the map transformation was able to preserved the partitioner in the result RDD, it may no longer make sense because maybe the keys are now different. So if I have some partitioner that make sense,therefore if the map transformation was able to preserve the partitioner in the resulting RDD, it may no longer make sense because now the keys maybe different. So they will no longer correspond to the correct partitions any more, that should make sense with this doh thing right? Because if I had some keys that made sense before then I would just replace them with the doh. Then what? Now the partitioner doesn't

correspond to the keys that I have, this is why the mapValues operation is so important. And this is why you should generally try to use it if you can, so if you're ever working with a pair RDD, you should try to first reach for mapValues. This operation makes it possible for us to still do map transformations While making it impossible for us to change the keys in a pair of RDD. Thus making it possible for us to keep the same partition around. That's clever, right. So, if we do all of these work to intelligently partition our data across our cluster, we can still do a map operation. And we can still keep the partition around that corresponds to how our data is organized by using the map values function. That's great, in the next session we'll continue learning about how partitioning can significantly reduce data shuffling. And how it can be a really huge boon for performance if tuned carefully.

Optimizing with Partitioners



In this session, we'll dig a little bit deeper into partitioners and we'll cover why someone might want to use a partitioner. So far we've seen the different sorts of partitioners, but we've not seen why they should really be used.

Optimizing with Partitioners

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

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

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

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

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

In the last session, we saw that Spark makes a few different kinds of partitioners available out-of-the-box to users. These were hash partitioners and range partitioners. And we got a little bit of a sense of when one might be used versus the other. We also looked at the sorts of operations that either may introduce new partitioners, or which may carry long unexisting partitioner. Or which could even completely discard a partitioner. Said simply, the biggest reason why someone would want to care about partitioning is because partitioning can bring enormous performance gains, especially in the face of operations that may cause shuffles. The basic intuition is that if you can somehow optimize for data locality, then you can prevent a lot of network traffic from even happening. We know that network traffic means huge latencies. So partitioning can mean significantly reduced latencies which, of course, translates into better performance.

Optimization using range partitioning

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

```
val pairs = purchasesRdd.map(p => (p.customerId, p.price))
val tunedPartitioner = new RangePartitioner(8, pairs)

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

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

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

As usual, let's look at an example to see how much partitioning can actually help performance. Let's return back to this groupByKey, which is rreduceByKey example that we saw in previous sessions. When we left off with this example, we were pretty happy with the 3X performance boost for using reduceByKey instead of groupByKey. However, we can still do better than that. You might not believe it, but we can use range partitioners and we can optimize our earlier use of reduceByKey. So it doesn't even have to involve shuffling over that work at all, here's how we do it. Although what we've done differently in this example is that we've started by partitioning our initial dataset. So all of this right here, this is still the same from before. So we start by creating a simple range partitioner here. It has 8 partitions, because, perhaps, this number of partitions makes sense for our cluster. And then we pass into our partitionBy function, this RangePartitioner that we just created. And finally we persist() this newly partitioned RDD. Then we just do the same thing that we did previously, we make a pair RDD and then we use the reduceByKey() and that's it. Seems simple enough, but how does it compare when we run it on the cluster?

Optimization using range partitioning

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

purchasesPerMonthSlowLarge: Long = 100000
Command took 15.48s
```

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

purchasesPerMonthFastLarge: Long = 100000
Command took 4.65s
```

On the range partitioned data:

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

purchasesPerMonthFasterLarge: Long = 100000
Command took 1.79s
```

almost a 9x speedup over purchasePerMonthSlowLarge!

Well, it's significantly faster, this is the result here. We have almost a 9x speed up over the groupByKey version. So 9x speed up over this one here. So imagine instead that these computations weren't running in seconds, but they were running in minutes or hours. This 9x speed up is a pretty big difference then.

Partitioning Data: partitionBy, Another Example

From pages 61-64 of the Learning Spark book

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

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

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

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

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

Okay, so let's look at another example. This one comes from the Learning Spark book, which is a really great book. And you can find this example on pages 61 to 64 of the Learning Spark book. So let's

imagine` that we have an application. And let's say that this application is a media application where users can, for example, subscribe to topics for articles. They want to read articles, so they say I'm interested in a certain topic. And let's imagine that in this application, we have a huge table of user information. And we store that big table in memory in the form of a pair RDD. We can call these big datasets userData here, and since it's a pair RDD, its key is a UserID and its value is UserInfo where UserInfo contains a list of topics that the specific users subscribe to. Now imagine, every now and then this application has to combine this big dataset of userData with a smaller dataset representing events that have happened in the last five minutes. We'll call the smaller dataset events here, and it too has key User ID. But its value instead is of type LinkInfo here. Which represents which links that that user has clicked on in the past five minutes. And in this example we may wish to count how many users visited a link that was not related to one of the topics that they've subscribed to. So this is kind of reminiscent of the CFF application that we saw in previous sessions. We know that we're going to have to join these two datasets somehow. Because we're going to want to group the UserInfo to the LinkInfo for those users that have been active in the past five minutes. So what might this program look like?

Partitioning Data: partitionBy, Another Example

From pages 61-64 of the Learning Spark book

```
val sc = new SparkContext(...)
val userData = sc.sequenceFile[UserID, UserInfo]("hdfs://...").persist()

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

Is this OK?

Well here's the full thing implemented on one slide. Of course, the most important part is this line here, where we take the userData and then we do an inner join with events. Now we have a new parity called joined, which contains all the users that have been active in the past five minutes and what they've been clicking on. Now all we have to do is filter that dataset down to figure out if people are clicking on things that they're not subscribed to. So we do that here, and finally, we just count it up to see what that number is. So what do you think about this program? Does it seem all right? Logically it seems like it should do what we want it to do, right?

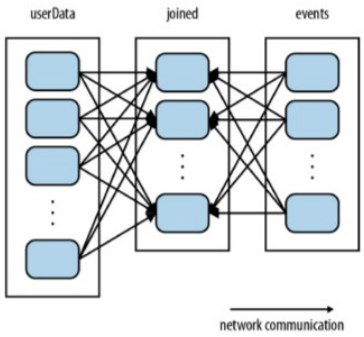
Partitioning Data: partitionBy, Another Example

From pages 61-64 of the Learning Spark book

It will be very inefficient!

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

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



Well, although, it might do the right thing logically, it'll be really inefficient. This is because each time this function here, this processNewLogs function is invoked. The join has no idea about how any of the keys are partitioned in these datasets. And so all that Spark can do, is hash all of the keys of both datasets. And then send the elements with the same key hash across the network to be on the same machine. Then once that's done, it can join together the elements with the same key on that machine. And it does this every time this method is invoked, even though this really huge pair RDD called userData doesn't change. So there's no reason to keep sending this thing around if it's always the same, right? It shouldn't have to keep moving all this data around the network. To give you some sense of what this looks like, here's a little diagram. This is the userData dataset, and then you have the joined dataset here. Basically what's happening is that we have this big shuffle between userData and joined.

Partitioning Data: partitionBy, Another Example

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

Therefore, userData becomes:

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

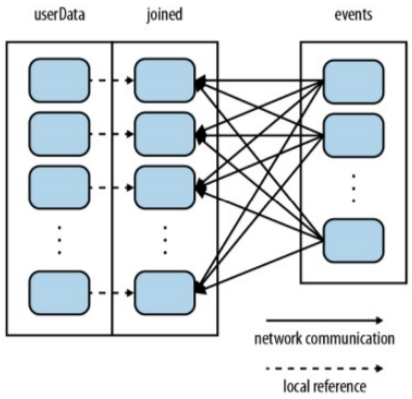
Since we called partitionBy when building userData, Spark will now know that it is hash-partitioned, and calls to join on it will take advantage of this information.

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

While it might not seem like it, fixing this is actually quite easy. As you might have guessed, all we have to do is partition the really big userData RDD right at the beginning of the project. So you remember here, we create the userData RDD. So all we have to do is add these two lines here to userData. That is, we just call partitionBy, and we pass to it a new HashPartitioner, with some fixed number of partitions. In this case, we say we want 100 partitions. And after that, all we have to do is just persist() the userData. So, since we called partitionBy while we were building up user data, Spark now knows that it's hash partitioned. And therefore calls to join on this userData thing here, can take advantage of this partitioning. So now, when we call userData join(events), like this here, when we do userData join(events) like we saw on the original program. Spark only has to shuffle the event RDD sending events with specific user IDs to the machine that contains a corresponding hash partition of user data. So, The bottom line is that the tiny pair RDD event, the event one, the small one should now be the one that's shuffled. And not the big RDD full of user data.

Partitioning Data: partitionBy, Another Example

Or, shown visually:



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

Or shown more visually, now that the userData pair RDD is pre-partitioned. Spark only has to shuffle the events RDD. This means that events with specific user IDs are the only ones that have to be sent to machines with corresponding hash partitions of user data. No more shuffling that big userData RDD.

Back to shuffling

Recall our example using groupByKey:

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

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

Grouping is done using a hash partitioner with default parameters.

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

Now, let's look back to our example, using groupByKey. Let's try to understand a little bit about what's happening with these partitioners under the hood. We recall that grouping all the values of the key-value pairs with the same key requires collecting all of the key-value pairs with the same key on the same machine. That's a mouth full, by default, grouping is done using a hash partition with the default parameters. And then the resulting RDD is configured to use that same hash partitioner that was used to construct it. This is what's going on under the hood.

How do I know a shuffle will occur?

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

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

At this point, you might be asking yourself, well, goodness, how do I know when a shuffle will occur? Well, there's a simple rule of thumb to try and remember to determine when a shuffle might occur. A shuffle can occur when a resulting RDD depends on other elements from either the same RDD or another RDD. So certain operations like join just giving how they fundamentally work. The idea is that they should be depending on data from other parts of the same RDD or other RDDs. This should, by default, in your mind trigger some sort of warning that, aha, shuffling could occur. What can I do to reduce it? And, of course, the answer is that if you intelligently partition your data you can either greatly reduce or completely prevent shuffling from occurring.

How do I know a shuffle will occur?

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

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

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

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

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

There are also other tricks and even methods that can help you figure out when a shuffle has been either planned or executed. On the one hand, you could look at the return type of certain transformations like we saw earlier in this session. Sometimes you might see a return type called ShuffledRDD here. Of course, this is evidence that a shuffle is either already happened is planned. There's also this very handy method called toDebugString that you can call an RDD to see it's execution plan. This will give all kinds of information about how your job is planned. And you can keep an eye out for warning signs like the ShuffleRDDs here.

Operations that *might* cause a shuffle

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

But perhaps the best thing to do is just keep in mind which operations might cause a shuffle. You can usually tell by logically thinking through what the method does. In any case, here's a handy list of operations that might cause a shuffle. Of course, the usual suspects are here. We have the different kinds of joins. We have the groupByKey, reduceByKey, combineByKey. And then there are other operations like repartition or coalesce. And as their names suggest, these operations sound like they have something to do with partitioning. So, of course, if they are partitioning data, they're going to be moving data around. So these are the sorts of operations that may possibly cause a shuffle.

Avoiding a Network Shuffle By Partitioning

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

Can you think of an example?

2 Examples:

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

However, if I've hammered anything into your head during this session. The one thing that you should try to remember is that it is possible to sometimes avoid a network shuffle by partitioning. There are two common scenarios where you can avoid network shuffling by partitioning. The first is when you use an operation like groupByKey on pre-partitioned RDDs. So this example, here, this causes values

to be computed locally since they've already been pre-partitioned, or preshuffled, so to speak. So, the work can all be done on the local partitions on the worker nodes, without workers having to shuffle their data again to one another. And, in this case, the only time the data has to be moved is when the final reduce values have to be sent back from the worker nodes to the driver node. The other common scenario here has to do with pre-partitioning before doing joins. So, we can completely avoid shuffling by pre-partitioning the two joined RDDs with the same partitioner. Of course, you must also ensure that the pre-partitioned RDDs are cached following the partitioning. Of course, this makes it possible to compute the entire join locally without any network shuffling. Since the data that must be joined together from both pair RDDs has already been relocated to live on the same node in the same partition. So, you don't need to move the data around in this case.

Shuffles Happen: Key Takeaways

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

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

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

So you take anything away from the session, you should remember that how you organize your data on your cluster really matters. You can go from repeatedly shuffling large datasets while trying to join it with a smaller dataset to not having to shuffle any data at all. All just by organizing and partitioning your data intelligently from the beginning of your job. And remember that we saw speedups of up to around 10x on small examples. By just trying to ensure that data isn't transmitted over the network to other machines, when it doesn't have to be. If we think back through the latency numbers that we learned in the first week, you should have the intuition that this 9, 10x speedup could make a big difference in your day to day work. If you're trying to run a job that should complete in 4 hours, but you miss an opportunity to partition data or optimize away a shuffle. It could take 40 hours instead, so this is why partitioning is important.

Wide vs Narrow Dependencies

Wide vs Narrow Dependencies

Big Data Analysis with Scala and Spark

Heather Miller

In this session,we're going to focus on wide versus narrow dependencies, which dictate relationships between RDDs in graphs of computation, which we'll see has a lot to do with shuffling.

Not All Transformations are Created Equal

Some transformations significantly more expensive (latency) than others

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

In the past sessions:

- we learned that shuffling sometimes happens on some transformations.

In this session:

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

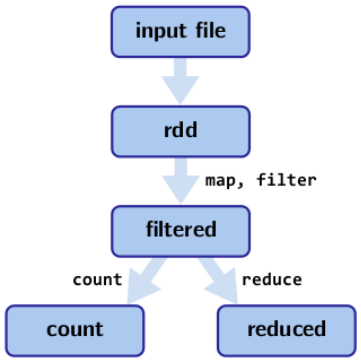
A recurring theme lately has been that, not all transformations are exactly equal. Some are a lot more expensive in terms of latency than others. For example, some might require lots of data to be transferred over the network perhaps unnecessarily so, operations that cause a shuffle are an example of this. In order to better understand when shuffling might occur, we have to look at how RDDs are represented, and then we have to think carefully about the operations that we're calling on our RDDs, to understand when a shuffle might occur. We'll actually see that this unique way of representing RDDs, actually makes fault tolerance possible in Spark. So we mentioned in earlier sessions that Spark is fault tolerant, we'll get a glimpse of how that's possible and actually kind of easy.

Lineages

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

Example:

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



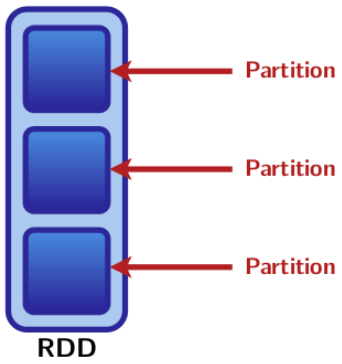
Spark represents RDDs in terms of these lineage graphs/DAGs

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

So let's start with some terminology. Let's start with lineages. So if you think about a group of computations that are done on a RDD, we can call that group of computations, a lineage graph. That is when you do operations on RDD, the operations can be organized into a Directed Acyclic Graph, representing the computations that were done on that RDD. So here's an example, we start with some input RDD, then we call the number of operations on that RDD. We can then reuse one of these RDD's in two subsequent computations. So we can reuse this filtered RDD in account, and we can also reuse it in a reduce. If you try to visualize it, it makes a graph like this. An rdd is the rdd that's actually made from this input file. We can then do map and filter, these transformations here are not rdd. And we'll get back in new RDD called filter, which we could persist in memory. And then we can use that filtered RDD in two different ways, with a count and with a reduce operation. This forms a graph of computations of tree without cycles. That's why it's called a Directed Acyclic Graph. In Spark, these graphs or DAGs are referred to as the lineage graph. That means it's possible to step back up this graph, to figure out how the result of this count operation is derived from input file. These lineage graphs are a big part of how RDDs are represented in Spark. And in fact, Spark can actually analyze this representation and do optimizations. We'll see what that means in a moment.

How are RDDs represented?

RDDs are made up of 2 important parts.
(but are made up of 4 parts in total)

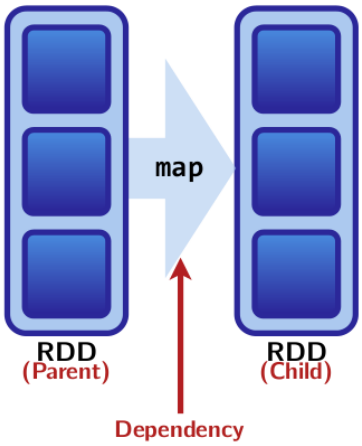


RDDs are represented as:

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

How are RDDs represented?

RDDs are made up of 2 important parts.
(but are made up of 4 parts in total)

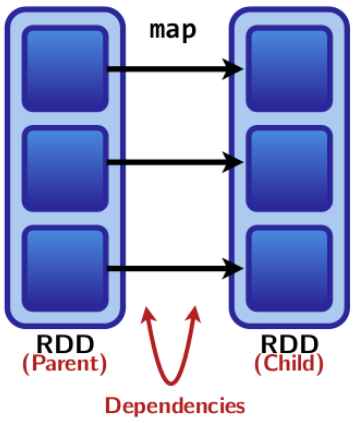


RDDs are represented as:

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

How are RDDs represented?

RDDs are made up of 2 important parts.
(but are made up of 4 parts in total)

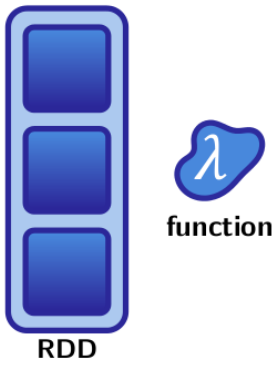


RDDs are represented as:

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

How are RDDs represented?

RDDs are made up of 2 important parts.
(but are made up of 4 parts in total)



RDDs are represented as:

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

But first, let's look at how RDDs are represented. So let's start with a visual example. Let's assume that this block here is an RDD. Usually, we think of RDDs as some kind of abstraction that we invoke methods on. But an RDD itself is actually made up of Partitions, partitions are atomic pieces of the dataset, and they may exist on one or many compute node. So you might have one RDD spread out over many compute nodes, and all of the pieces that are spread out are the Partitions. The next most important part of an RDD are Dependencies. Dependencies model relationship between an RDDs partitions, and the partitions that it was derived from in other RDDs. So for example in this picture, if this is a parent RDD and this is a child RDD, and we're calling the map function on this RDD to get that RDD. Then the Dependencies are how these partitions mapped to those partitions with the map function. So, while you might normally think of a map as one big arrow between this RDD and that

RDD, actually Dependencies model the relationship between some partitions and the partitions that they were derived from. So you could say in this picture that this partition was derived from that partition. This Partition was derived from that partition and so on. And finally the last two parts of an RDD include a function, so you can think of this as the function that you might you might pass to a map operation on the previous slide, you can't compute a child partition without some kind of function to get that child partition from the parent partition. So functions are actually a big part of RDDs, because they say how to compute the data sets based on an RDDs parents. So what to actually do. And finally, there is some metadata about the partitioning scheme and where data is placed that's a part of an RDD. So these are the four parts of an RDD.

RDD Dependencies and Shuffles

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

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

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

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

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

But you might ask well how does that relate to shuffles? If you recall in a previous session, we developed the following rule of thumb. We said a shuffle can occur, when the resulting RDD depends on other elements from the same RDD or another RDD. So here's a key word, depends. Now we have an idea of what a dependency is, in fact an RDD's dependencies actually encode when data must move over the network. So that brings us to transformations. So we know that transformations cause shuffles, and we know that dependency information can tell us when a shuffle might occur. To differentiate between these, we can actually define two sets of dependencies called Narrow and Wide Dependencies, that can tell us when shuffles will happen. We're going to go more into depth about what a Narrow Dependency and a Wide Dependency means in this session.

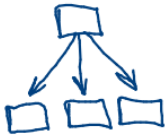
Narrow Dependencies vs Wide Dependences

Narrow Dependencies

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

Wide Dependencies

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



Narrow Dependencies vs Wide Dependences

Narrow Dependencies

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

Fast! No shuffle necessary. Optimizations like pipelining possible.

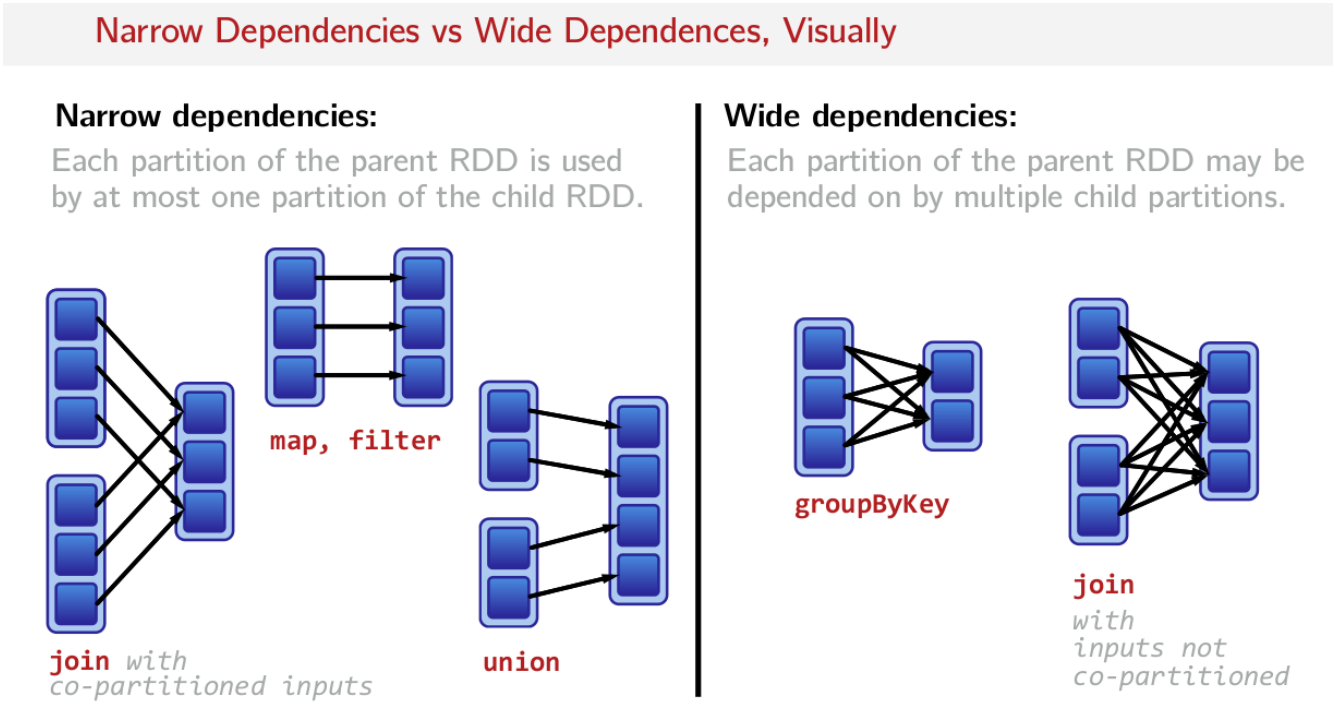
Wide Dependencies

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

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

So let's define Marrow and Wide Dependencies. A transformation has Narrow Dependencies, when each partition of the parent RDD is used by at most one partition of the child RDD. So that means some child partition has only one parent partition and not many. A transformation that has Wide Dependencies on the other hand, is one where each partition of the parent RDD may be depended on by multiple children partitions. So that means you may have many child partitions, which were all derived from a single parent partition. Transformations with these kind of dependencies have Wide Dependencies. So what does that mean to us? Well, as you might've guessed, transformations with Narrow Dependencies are typically quite fast. They require no shuffles, and optimizations like pipelining can occur. Which is when you can group together many transformations into one pass. And as you may have guessed, transformations with Wide Dependencies are slow, because they require all

or some data to be shuffled over the network. So these are the transformations that cause shuffles.

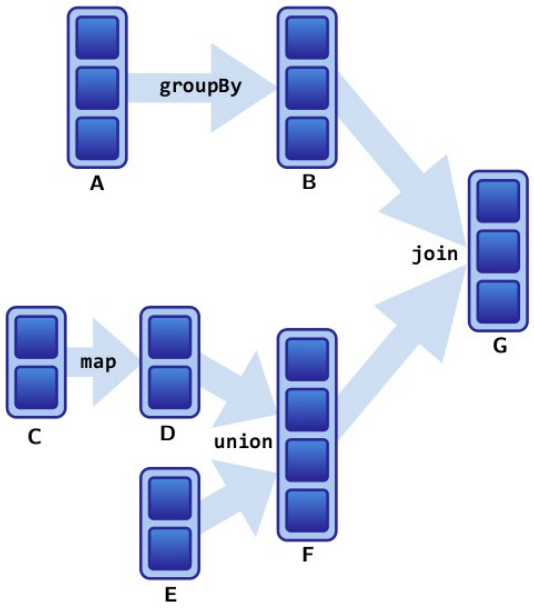


So let's try to visualize the differences between Narrow and Wide Dependencies. Let's start first with Narrow Dependencies. So as I said on the previous slide, each partition of the parent RDD is used by at most one partition of the child RDD. So some example transformations include map and filter, because one partition depends on at most one other partition. And the same is true for union. We can essentially just put the data in this partition, into the new resulting RDD here. There's no relationship between many RDDs, when it comes to looking at who this partition was derived from. And what might surprise you, is that this is also true for joins that are already partitioned. So if we recall in previous sessions, how we actually were able to create joins where shuffles weren't required, this is the sort of join which has a Narrow Dependency. But just to go back to the definition of a Narrow Dependency, join with co-partitioned inputs, is a transformation with narrow dependencies. Because each partition of the parent RDD, is used but at most one partition of the child RDD. That is this child dependency here, is derived from only one partition in one of the parent RDDs. So in another way, we don't have an error going from this partition to that child partition, and then also this partition here going to that same first child partition. We have just one relationship, this partition relates to that partition and that's it. On the other hand, transformation with Wide Dependencies as we said on the previous slide, is when each partition of a parent RDD may be depended on by multiple child partitions. So one operation that we already know caused the shuffles is groupByKey. Now when we visualize groupByKey with partitions and dependencies, we can see how a shuffle will occur. As the definition here says, we have parent partitions which are depended on by multiple child partitions. So for example, this partition is depended on by both of the children partitions. So these are Wide Dependencies. Another operation with Wide Dependencies of course is join, when it's inputs are not already partitioned, this means that we are going to have to move data all over the network to make sure that the resulting RDD has all the values corresponding to some key on the same partitions. Before the join is called, they could be spread all over the network. So this is a pretty clear example of an operation that has Wide Dependencies, a join without already partitioned inputs.

Narrow Dependencies vs Wide Dependences, Visually

Let's visualize an example program and its dependencies.

Conceptually assuming the DAG:



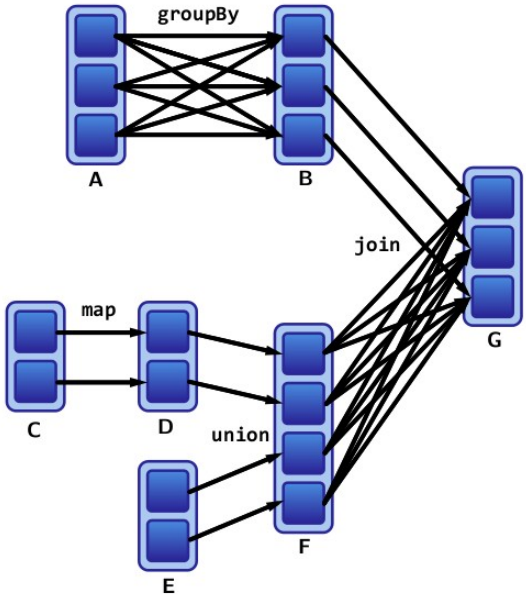
What do the dependencies look like?

Which dependencies are wide, and which are narrow?

So let's take this visualization a step further, and let's visualize a sample program and its dependencies. So let's assume we have the following DAG, the following graph of computations. Where each one of these here are RDDs. So we have this box is an RDD, this one's called A, this one's called B, this one's called C, D, E, F, G. They each have different numbers of partitions. So A has three partitions, C has two partitions. And these big arrows here, represent the operations that we're calling on those RDDs. So A is the parent RDD. And B is the child RDD, and the operation that B is derived from is groupBy. So here's a quick exercise for you. If you had to draw the dependencies between each of these RDDs, what would they look like? Which dependencies are wide and which dependencies are narrow? Try it for yourself. Try to draw it out on a piece of paper, and think very carefully about what functions like groupBy and map, actually semantically do. Ask yourself how many dependencies each one of these partitions might have on their parents.

Narrow Dependencies vs Wide Dependences, Visually

Let's visualize an example program and its dependencies.

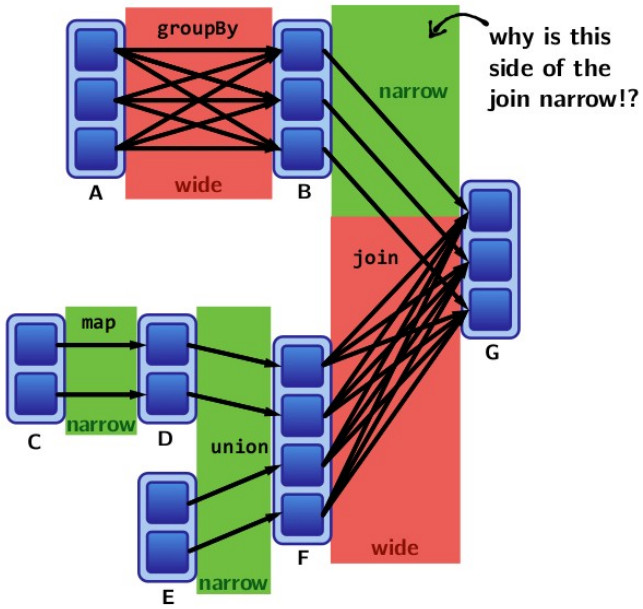


Which dependencies are wide, and which are narrow?

Well, here's the answer. If you think very carefully about what groupBy has to do, it has to group values by the same key on in the resulting RDDs. So it has to move key value pairs around the network, in order to make sure that they're all in the same partition. Which means that many of the parent RDDs partitions, are used in each partition of the child RDD. Operations like map have only one dependency. So this partition depends on only that partition and it doesn't matter what's in this partition.

Narrow Dependencies vs Wide Dependences, Visually

Let's visualize an example program and its dependencies.



Wide transformations:
`groupBy, join`

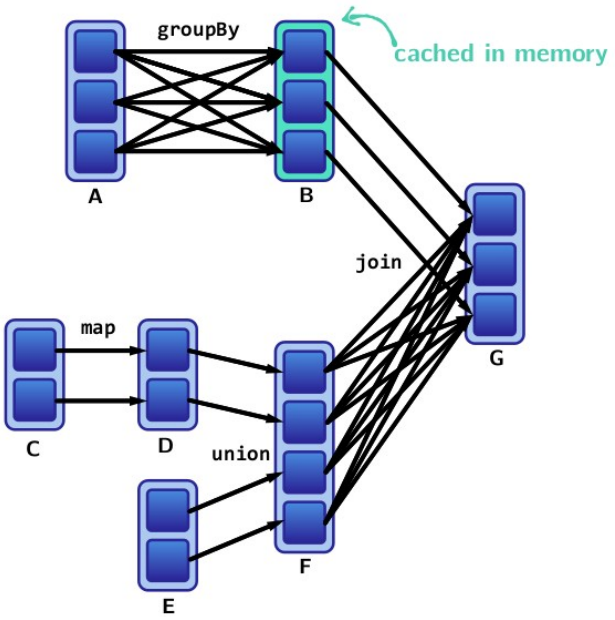
Narrow transformations:
`map, union, join`

Narrow Dependencies vs Wide Dependences, Visually

Let's visualize an example program and its dependencies.

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

Part of this join is thus a narrow transformation.



So the next question I asked you, are which dependencies are wide and which are narrow? Transformations like groupBy and join are wide, because each child partition depends on multiple parent partitions. And transformations like map and union are narrow, because each child depends on only one parent RDD. There's a one-to-one relationship between parent partitions and child partitions. Well, hang on a second. You might be asking yourself, well, why is this side of the join narrow? And the simple answer is that if you looked at the code, you probably cached B in memory because you knew you were doing a group by key, which does partitioning already. So you want to keep that partitioning in place when you reuse B. So I said simply, B then is already co-partitioned. That's already partitioning in cached memory following the call to this groupBy here. That's why this part of the join can have Narrow Dependencies.

Which transformations have which kind of dependency?

Transformations with narrow dependencies:

`map`
`mapValues`
`flatMap`
`filter`
`mapPartitions`
`mapPartitionsWithIndex`

Transformations with wide dependencies:

(might cause a shuffle)

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

So I hope those visualizations were an aha moment for you, when thinking about computations that you have to do in Spark. You actually have to think carefully about whether or not an operation that you want to use, might have Narrow or Wide Dependencies. And of course we saw that sometimes some operations like join, sometimes are narrow and sometimes are wide. So, I can make a list here of operations that are usually always narrow and usually always wide. Though there of course some exemptions such us join here, because we know that join is not always wide. Sometimes it can be narrow depending on the partitioning scheme. But here we can roughly breakdown transformations into narrow and wide. Again, doesn't always hold but it's a rough approximation. So you can always look back at this list and use it as little bit of our reference, but it's not always going to give you the answer. You're still going to have to think carefully about the operation that you're actually doing on the cluster.

How can I find out?

`dependencies` method on RDDs.

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

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

Narrow dependency objects:

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

Wide dependency objects:

- `ShuffleDependency`

But sometimes, maybe you want to be a little bit more certain than thinking critically about the operations that you're using. For that, Sparks gives you a method called dependencies. And dependencies will give you a sequence of dependency objects, which are actually the dependencies that Spark uses in its scheduler, to know how this RDD depends on other RDDs. When you call this method, you'll get back a sequence of dependency objects with different names. Dependencies called OneToOneDependencies are Narrow Dependencies. You might also see something called a Prune and a Range Dependency. But most often you'll see OneToOneDependency for narrow transformations. And finally wide dependencies are called ShuffleDepedency in Spark, which is a pretty clear indicator that a shuffle might occur. Here's a quick example of what the output to dependencies looks like. So if you have some pair RDD words, and you do groupByKey and then you call dependencies. What you get back from the Spark is a list with shuffle dependencies in it. So you know that this called to groupByKey is going to cause the shuffle.

How can I find out?

dependencies method on RDDs.

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

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

How can I find out?

toDebugString method on RDDs.

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

```
val wordsRdd = sc.parallelize(largeList)
val pairs = wordsRdd.map(c => (c, 1))
                    .groupByKey()
                    .toDebugString
//pairs: String =
//(8) ShuffledRDD[219] at groupByKey at <console>:38 []
// +-(8) MapPartitionsRDD[218] at map at <console>:37 []
//    |  ParallelCollectionRDD[217] at parallelize at <console>:36 []
```

Spark also comes with another method called toDebugString. This method actually prints out a visualization of the RDD lineage along with other information relevant to scheduling. So if I was to use the same example on the previous slide, so we do map and then groupByKey on some pair RDD of words, if we call toDebugString, what we see is this. The resulting RDD from group by key is a ShuffledRDD, which came from a MapPartitionsRDD. Which itself came from a ParallelCollectionRDD. So this ParallelCollectionsRDD corresponds to wordsRdd. This MapPartitionsRDD corresponds to the results of map on this wordsRdd. And finally, ShuffledRDD corresponds to the result of groupByKey. The indentations here actually show how Spark groups together these operations. So operations in the same indentation, like for example MapPartitions here, these groupings are typically separated by shuffles. So you can print out a lineage using toDebugString, and you'll be able to see how your job will be broken up into different groupings of operations, separated by shuffles. So for example you can have a group of many narrow transformations, followed by one wide transformation. And you can see that from the indentation from the lineage printed out,

from toDebugString. And if you want to understand a little bit more about how Spark Jobs are actually run, these groupings are called stages. So a spark job is broken into stages by its scheduler.

Lineages and Fault Tolerance

Lineages graphs are the key to fault tolerance in Spark.

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

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

So we saw these lineage graphs and how they actually represent all of the relationships between RDDS, and how we can determine when a shuffle will occur, but at the beginning of this session, I made this claim that lineage is related to fault tolerance. So how might that be the case? Well, actually lineages are the key to fault tolerance in Spark. Ideas for functional programming enable this fault tolerance in Spark. This is because these are immutable, so we can't update or change any of the data inside of an RDD. And then by using higher-order functions like map, flatMap and filter to do functional transformations on our immutable data, what we have is effectively a Directed Acyclic Graph. And assuming that we keep around this function that we passed to operations like map, flatMap and filter, what we can do is we can re-compute at any point in time any of the RDDs in our entire lineage graph. So with these three aspects of Spark's design, RDDs being immutable, everything being essentially higher order functions, and then passing functions to these higher order functions, and building a Directive Accyclic Graph, building a tree of transformation out of that.

Lineages and Fault Tolerance

Lineages graphs are the key to fault tolerance in Spark.

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

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

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

in-memory + fault tolerant!

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

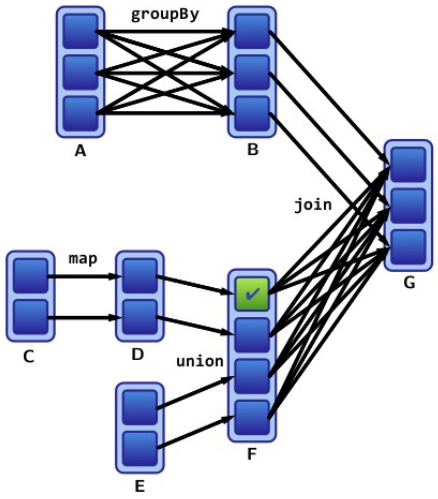
Recover from failures by recomputing lost partitions from lineage graphs.

This means that we can actually, recover from failures by recomputing lost partitions from the lineage graphs. So if you keep this information around. If you save this somewhere to stable storage, then you'll always be able to re-derive individual RDDs in your lineage graph. That's the key idea here. And that's how Spark achieves fault tolerance without having to write to disk. This is how Spark can still do everything in memory, and also be fault tolerant. Of course, this is always more easily understood visually.

Lineages and Fault Tolerance, Visually

Lineages graphs are the key to fault tolerance in Spark.

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

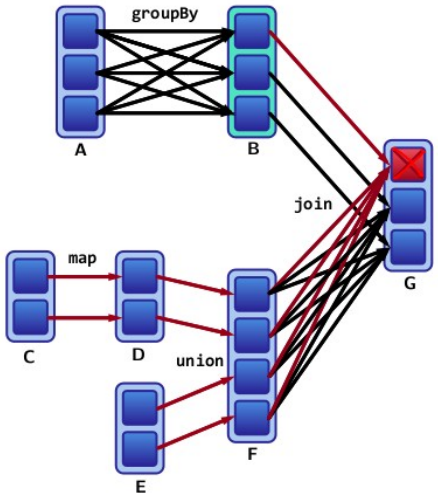


So let's assume one of our partitions from the previous example fails. So let's assume this partition here for some reason, is no longer any good. Without having to have checkpoints and all that data to disk, Spark can actually just re-drive it using this graph of dependencies that it all ready has. For example, we know that this piece of data is derived from this piece of data, which is derived from this piece of data. Assuming that non of it is cached in memory, and I can't just reuse this, and all we all have to do is recompute these pieces of data. So all we have to do is go back to this dependency information along with those functions that are stored with those dependencies, and just recompute these pieces of data in which we compute that piece of data, like so. And viola.

Lineages and Fault Tolerance, Visually

Lineages graphs are the key to fault tolerance in Spark.

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



Now note, now we can put in this partitions is fast for narrow dependencies, but it's slow for y dependencies. But looking then at a visualization to see why that's the case. If we lost this partition instead, we have to trace our dependencies backward. And because this child dependency depends on all of the parent dependencies in this union, we've gotta recompute all of these intermediate RDDs, in order to recompute this one piece of data here. Of course, it helps that this RDD is cached memory so we can just reuse this partition in, without having to compute this groupBy. But this is still going to be very expensive. So losing partitions that were derived from a transformation with wide dependencies, can be much slower.