Distributed Key-Value Pairs

Principles of Functional Programming

Heather Miller

What we've seen so far

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

What we've seen so far

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

Spark's Programming Model

- We saw that, at a glance, Spark looks like Scala collections
- However, interally, Spark behaves differently than Scala collections
  - Spark uses ***laziness*** to save time and memory
- We saw *transformations* and *actions*
- We saw caching and persistence (*i.e.*, cache in memory, save time!)
- We saw how the cluster topology comes into the programming model
- We got a sampling of Spark's key-value pairs (Pair RDDs)

Today...

1. Reduction operations in Spark vs Scala collections
2. More on Pair RDDs (key-value pairs)
3. We'll get a glimpse of what "shuffling" is, and why it hits performance (latency)

Reduction Operations

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

Which of these two were parallelizable?

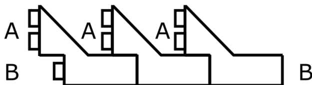
Reduction Operations

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

Which of these two were parallelizable?

`foldLeft` is not parallelizable.

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



Reduction Operations

`foldLeft` is not parallelizable.

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

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

Concretely, given:

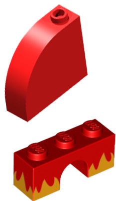
```
val xs = List(1, 2, 3)
val res = xs.foldLeft("")((str: String, i: Int) => str + i)
```

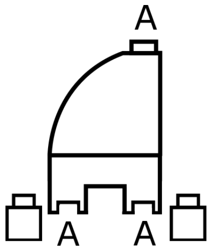
What happens if we try to break this collection in two and parallelize?

Reduction Operations: Fold

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

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



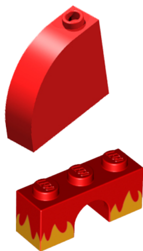


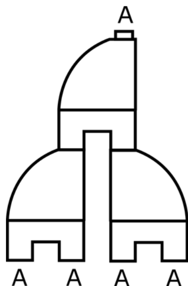
It enables us to parallelize using a single function `f` by enabling us to build parallelizable reduce trees.

Reduction Operations: Fold

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

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





Reduction Operations: Aggregate

Does anyone remember what `aggregate` does?

Reduction Operations: Aggregate

Does anyone remember what `aggregate` does?

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

Reduction Operations: Aggregate

Does anyone remember what `aggregate` does?

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

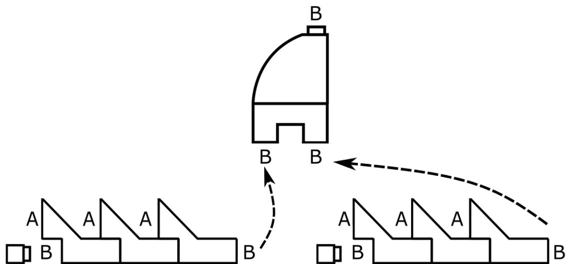
`aggregate` is said to be general because it gets you the best of both worlds.

Properties of `aggregate`

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

Reduction Operations: Aggregate

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



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

Reduction Operations on RDDs

Scala collections:

fold
foldLeft/foldRight
reduce
aggregate

Spark:

fold
~~foldLeft/foldRight~~
reduce
aggregate

Reduction Operations on RDDs

Scala collections:
fold
foldLeft/foldRight
reduce
aggregate

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

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

Reduction Operations on RDDs

Scala collections:
fold
foldLeft/foldRight
reduce
aggregate

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

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

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

Reduction Operations on RDDs

Scala collections:
fold
foldLeft/foldRight
reduce
aggregate

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

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

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

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

RDD Reduction Operations: Aggregate

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

RDD Reduction Operations: Aggregate

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

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

RDD Reduction Operations: Aggregate

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

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

Example:

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

RDD Reduction Operations: Aggregate

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

Example:

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

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

Pair RDDs (Key-Value Pairs)

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

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

Pair RDDs (Key-Value Pairs)

Creating a Pair RDD

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

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



val pairRdd = ???
```

Pair RDDs (Key-Value Pairs)

Creating a Pair RDD

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

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

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

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

Some interesting Pair RDDs operations

Transformations

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

Action

- `countByKey`

Pair RDD Transformation: groupByKey

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

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

Example:

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

val groupedRdd = eventsRdd.groupByKey()
```

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

Pair RDD Transformation: groupByKey

Example:

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

val groupedRdd = eventsRdd.groupByKey()

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

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

(Note: all code available in notebooks on Databricks Cloud.)

Pair RDD Transformation: `reduceByKey`

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

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

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

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

val budgetsRdd = ...
```

Pair RDD Transformation: reduceByKey

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

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

val budgetsRdd = eventsRdd.reduceByKey(_+_)

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

(Note: all code available in "exercise1" notebook.)

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

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

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

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

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

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

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

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

val avgBudgets = ???
```

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

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

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

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

Joins

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

There are two kinds of joins:

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

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

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

Inner Joins (`join`)

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

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

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

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

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

val trackedCustomers = ???
```

Inner Joins (`join`)

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

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

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

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

Inner Joins (join)

Example continued with concrete data:

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

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

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

Inner Joins (join)

Example continued with concrete data:

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

What happened to customer 104?

Inner Joins (join)

Example continued with concrete data:

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

What happened to customer 104?

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

Outer Joins (`leftOuterJoin`, `rightOuterJoin`)

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

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

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

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

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

Which join do we use?

Outer Joins (`leftOuterJoin`, `rightOuterJoin`)

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

Which join do we use?

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

Outer Joins (leftOuterJoin, rightOuterJoin)

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

Which join do we use?

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

Outer Joins (leftOuterJoin, rightOuterJoin)

Example continued with concrete data:

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

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

Outer Joins (`leftOuterJoin`, `rightOuterJoin`)

We can do the converse using a `rightOuterJoin`.

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

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

Outer Joins (leftOuterJoin, rightOuterJoin)

Example continued with concrete data:

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

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

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

Resources to learn more operations on Pair RDDs

Book

The *Learning Spark* book is the most complete reference.

Free

- **Spark Programming Guide**
  http://spark.apache.org/docs/1.6.1/programming-guide.html
  Contains compact overview of Spark's programming model. Lists table of all *transformers* vs *accessors*, for example. However, doesn't go into a lot of detail about individual operations.
- **Spark API Docs**
  http://spark.apache.org/docs/1.6.1/api/scala/index.html
  Look for `class PairRDDFunctions` for all possible operations on Pair RDDs.

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

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

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

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

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

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

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

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

Shuffles Happen

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

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