Reduction Operations

Big Data Analysis with Scala and Spark

Heather Miller

What we've seen so far

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

What we've seen so far

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

Spark's Programming Model

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

Transformations to Actions

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

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

Transformations to Actions

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

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

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

Reduction Operations, Generally

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

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

Reduction Operations, Generally

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

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

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

Reduction Operations, Generally

Reduction Operations:

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

(rather than another collection)

Example:

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

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

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

Parallel Reduction Operations

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

Which of these two were parallelizable?

Parallel Reduction Operations

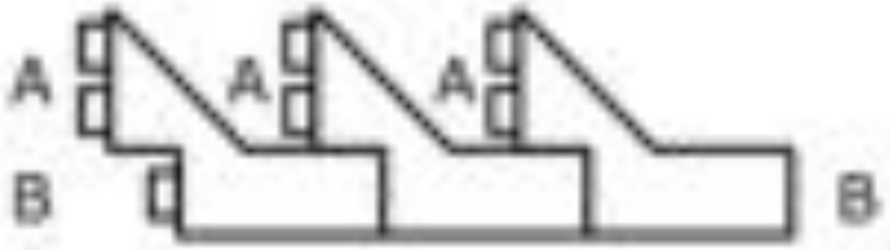
Recall what we learned in the course Parallel Programming course about `foldLeft` vs `fold`.

Which of these two were parallelizable?

`foldLeft` is not parallelizable.

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

> *Applies a binary operator to a start value and all elements of this collection or iterator, going left to right.*
>
> — Scala API documentation



Parallel Reduction Operations: FoldLeft

`foldLeft` is not parallelizable.

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

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

Concretely, given:

"1234"

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

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

Parallel Reduction Operations: FoldLeft

foldLeft is not parallelizable.

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

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

String

List(1,2)

"" + 1 → "1"
"1" + 2 → "12"
string

List(3,4)

"" + 3 → "3"
"3" + 4 → "34"
String

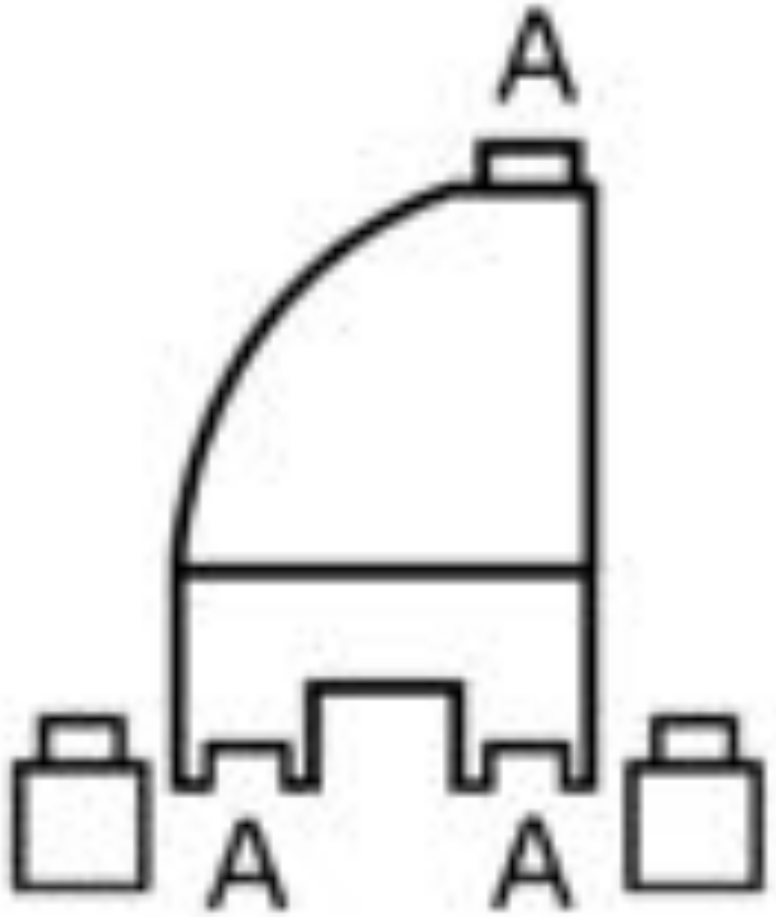
!! type error !!

can't apply
(str: String, i: Int) => str + i !!

Parallel Reduction Operations: Fold

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

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

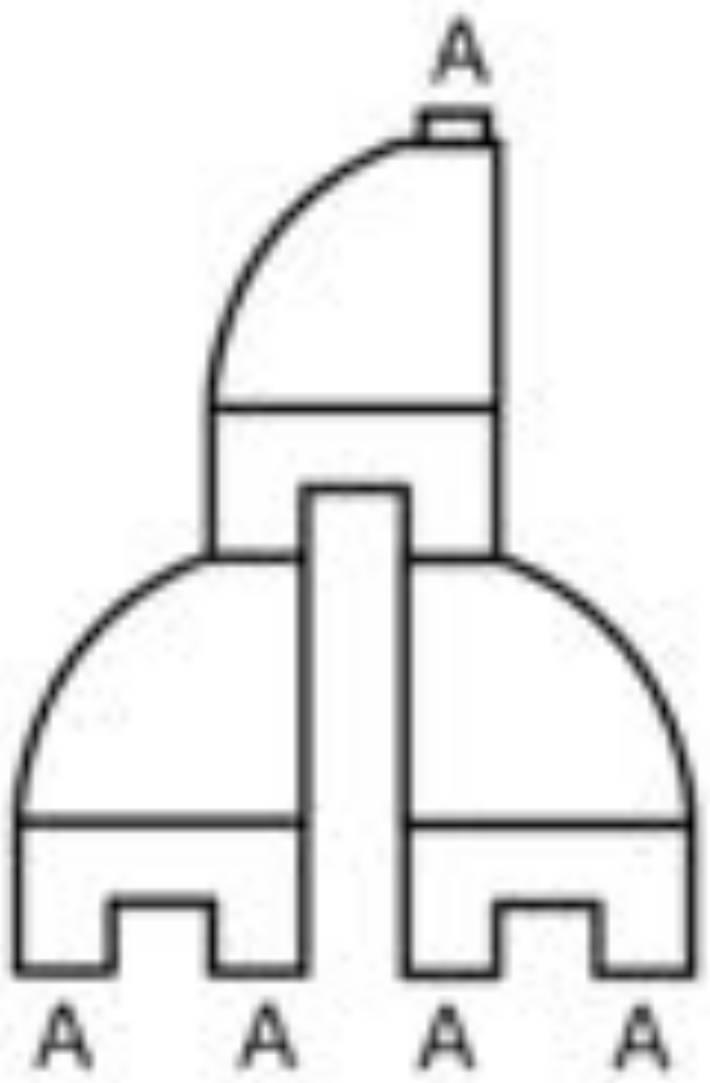


It enables us to parallelize using a single function `f` by enabling us to build parallelizable reduce trees.

Parallel Reduction Operations: Fold

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

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



Parallel Reduction Operations: Aggregate

Does anyone remember what `aggregate` does?

Parallel Reduction Operations: Aggregate

Does anyone remember what aggregate does?

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

Parallel Reduction Operations: Aggregate

Does anyone remember what `aggregate` does?

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

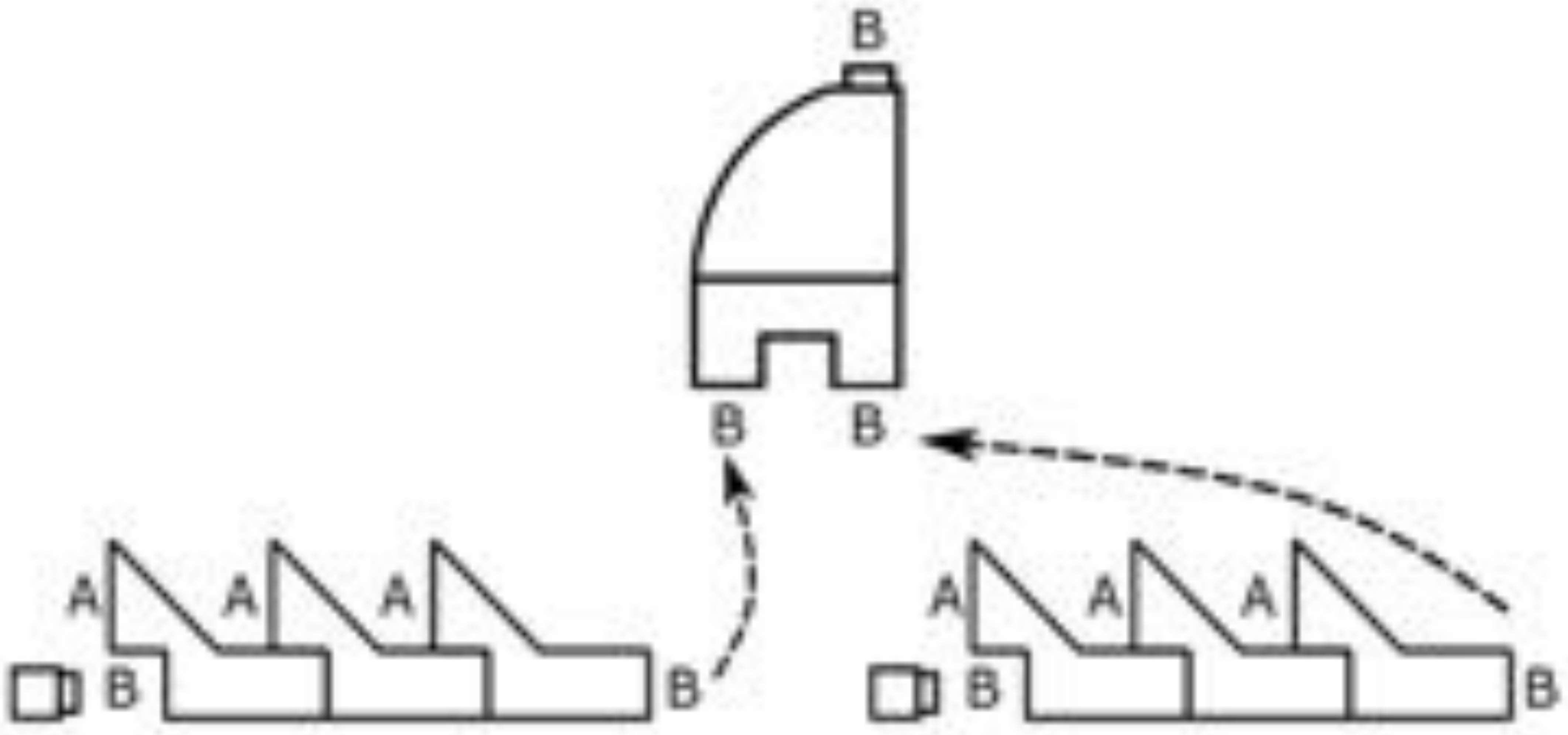
`aggregate` is said to be general because it gets you the best of both worlds.

Properties of `aggregate`

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

Parallel Reduction Operations: Aggregate

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



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

Reduction Operations on RDDs

Scala collections:
fold
foldLeft/foldRight
reduce
aggregate

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

Reduction Operations on RDDs

Scala collections:
fold
foldLeft/foldRight
reduce
aggregate

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

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

Reduction Operations on RDDs

Scala collections:
fold
foldLeft/foldRight
reduce
aggregate

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

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

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

Reduction Operations on RDDs

Scala collections:
fold
foldLeft/foldRight
reduce
aggregate

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

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

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

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

RDD Reduction Operations: Aggregate

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

RDD Reduction Operations: Aggregate

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

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

RDD Reduction Operations: Aggregate

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

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

Example:

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

RDD Reduction Operations: Aggregate

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

Example:

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

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

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

ÉCOLE POLYTECHNIQUE
FÉDÉRALE DE LAUSANNE

Distributed Key-Value Pairs (Pair RDDs)

Big Data Analysis with Scala and Spark

Heather Miller

Distributed Key-Value Pairs

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

Distributed Key-Value Pairs

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

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

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

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

Distributed Key-Value Pairs

(2004 research paper)

MapReduce: Simplified Data Processing on Large Clusters

Jeffrey Dean and Sanjay Ghemawat

jeff@google.com, sanjay@google.com

Google, Inc.

Abstract

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

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

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

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

Distributed Key-Value Pairs

(2004 research paper)

Abstract

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

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

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

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

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

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

Distributed Key-Value Pairs (Pair RDDs)

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

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

Distributed Key-Value Pairs (Pair RDDs)

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

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

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

Example:

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

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

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

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

Distributed Key-Value Pairs (Pair RDDs)

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

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

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

Distributed Key-Value Pairs (Pair RDDs)

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

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

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

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

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

Pair RDDs (Key-Value Pairs)

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

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

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

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

Pair RDDs (Key-Value Pairs)

Creating a Pair RDD

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

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



val pairRdd = ???
```

Transformations and Actions on Pair RDDs

Big Data Analysis with Scala and Spark

Heather Miller

Some interesting Pair RDDs operations

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

Transformations

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

Action

- `countByKey`

Pair RDD Transformation: `groupByKey`

Recall `groupBy` from Scala collections.

Pair RDD Transformation: `groupBy~~Key~~`

Recall `groupBy` from Scala collections.

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

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

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

Pair RDD Transformation: groupBy~~Key~~

Recall groupBy from Scala collections.

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

Example:

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

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

Pair RDD Transformation: groupByKey

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

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

Pair RDD Transformation: groupByKey

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

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

Example:

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

val groupedRdd = eventsRdd.groupByKey()
```

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

Pair RDD Transformation: groupByKey

Example:

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

val groupedRdd = eventsRdd.groupByKey()

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

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

Pair RDD Transformation: reduceByKey

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

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

Pair RDD Transformation: `reduceByKey`

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

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

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

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

val budgetsRdd = ...
```

Pair RDD Transformation: reduceByKey

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

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

val budgetsRdd = eventsRdd.reduceByKey(_+_)

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

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

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

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

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

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

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

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

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

Pair RDD Transformation: mapValues and Action: countByKey

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

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

.reduceByKey(

(org, budget) → (org, (budget, 1))

K V

Result should look like:

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

Pair RDD Transformation: mapValues and Action: countByKey

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

(budget, 1)

```
// 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))]
```

budgets

total # events

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

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

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

val avgBudgets = ???
```

Pair RDD Transformation: mapValues and Action: countByKey

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

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

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

Pair RDD Transformation: `keys`

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

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

Pair RDD Transformation: keys

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

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

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

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

val numUniqueVisits = ???
```

Pair RDD Transformation: keys

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

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

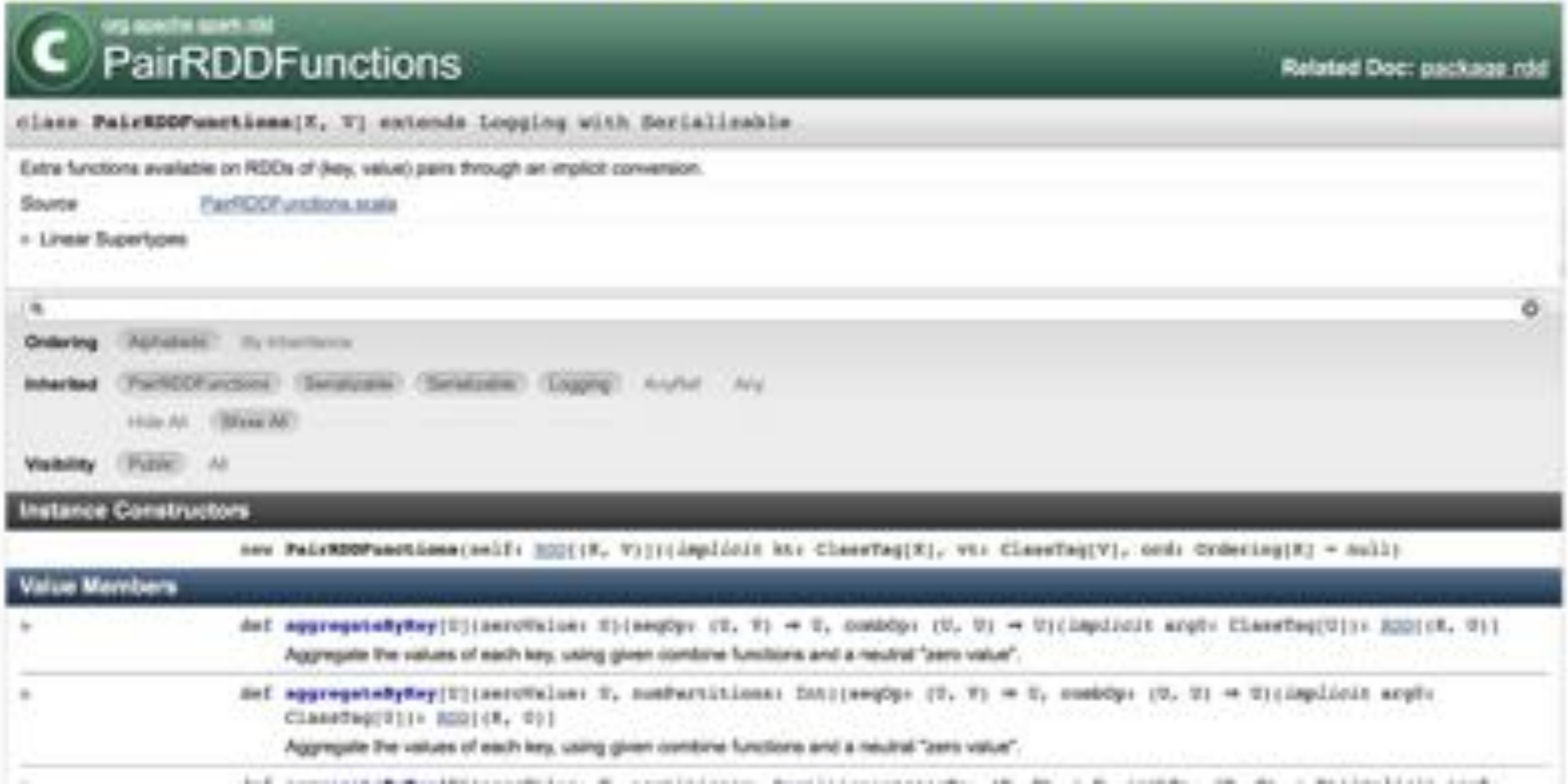
Example: we can count the number of unique visitors to a website using the keys transformation.

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

val numUniqueVisits = visits.keys.distinct().count()
// numUniqueVisits: Long = 3391
```

PairRDDFunctions

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



Pair RDDs (Key-Value Pairs)

Creating a Pair RDD

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

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

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

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

ÉCOLE POLYTECHNIQUE
FÉDÉRALE DE LAUSANNE

Joins

Big Data Analysis with Scala and Spark

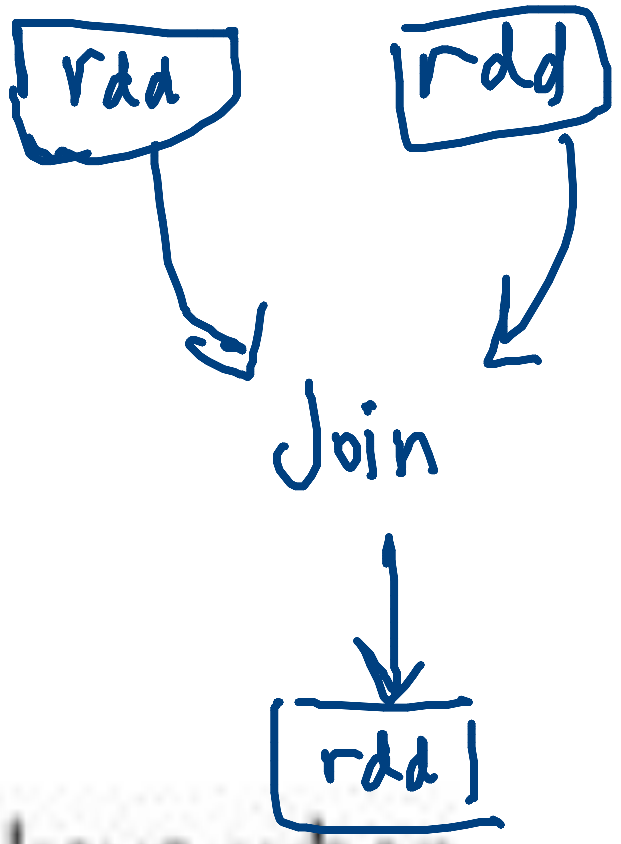
Heather Miller

Joins

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

There are two kinds of joins:

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



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

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

Example Dataset...

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

Let's assume the following concrete data:

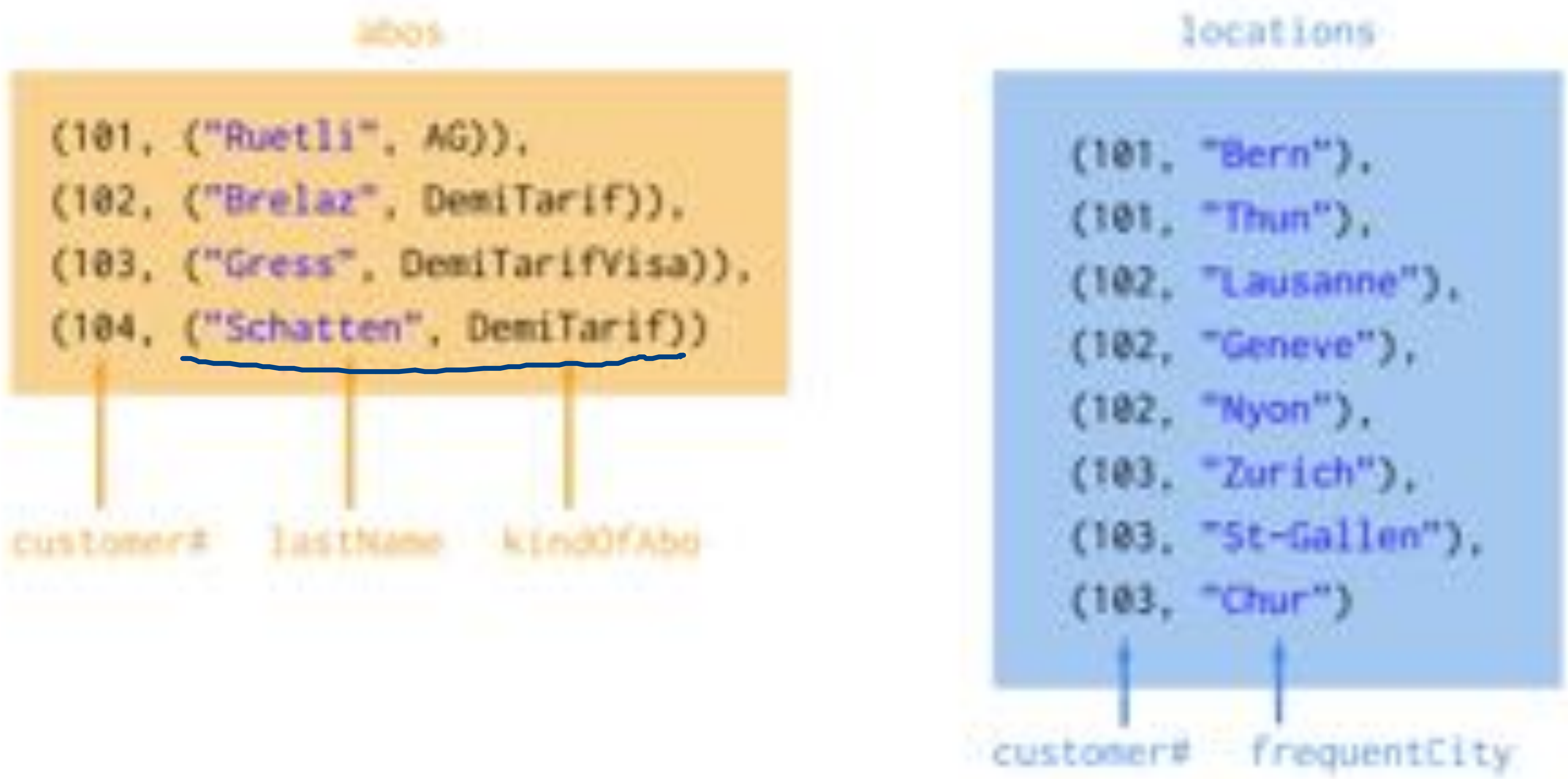
```
val as = List((101, ("Ruetli", AG)), (102, ("Brelaz", DemiTarif)),
              (103, ("Gress", DemiTarifVisa)), (104, ("Schatten", DemiTarif)))
val abos = sc.parallelize(as)

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

Example Dataset... (2)

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

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



Example Dataset... (3)

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

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

abos

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

This kind of data comes from CFF's databse of subscriptions

locations

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

This kind of data comes from individual purchases from the app (i.e., to use the app, you don't need an AG)

Inner Joins (join)

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

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

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

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

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

val trackedCustomers = ???
```

Inner Joins (join)

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

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

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

Inner Joins (join)

Example continued with concrete data:

abos

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

locations

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

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

Inner Joins (join)

Example continued with concrete data:

abos

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

locations

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

We want to combine both RDDs into one:

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

Inner Joins (join)

Example continued with concrete data:

abos

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

locations

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

We want to make a new RDD with only these!

Inner Joins (join)

Example continued with concrete data:

trackedCustomers

```
(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))
```

customer# lastName kindOfAbo frequentCity

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

Inner Joins (join)

Example continued with concrete data:

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

What happened to customer 104?

Inner Joins (join)

Example continued with concrete data:

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

What happened to customer 104?

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

Outer Joins (`leftOuterJoin`, `rightOuterJoin`)

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

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

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

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

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

Which join do we use?

Outer Joins (`leftOuterJoin`, `rightOuterJoin`)

Example continued with concrete data:

abos

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

locations

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

We want to combine both RDDs into one:

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

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

Outer Joins (leftOuterJoin, rightOuterJoin)

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

Which join do we use?

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

Outer Joins (`leftOuterJoin`, `rightOuterJoin`)

Example continued with concrete data:

abos

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

locations

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

We want to make a new RDD with these!

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

Outer Joins (leftOuterJoin, rightOuterJoin)

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

Which join do we use?

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

Outer Joins (leftOuterJoin, rightOuterJoin)

Example continued with concrete data:

abosWithOptionalLocations

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

customer# lastName kindOfAbo Option[frequentCity]

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

Outer Joins (leftOuterJoin, rightOuterJoin)

Example continued with concrete data:

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

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

Outer Joins (leftOuterJoin, rightOuterJoin)

We can do the converse using a rightOuterJoin.

abos

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

locations

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

We want to combine both RDDs into one:

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

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

Outer Joins (`leftOuterJoin`, `rightOuterJoin`)

We can do the converse using a `rightOuterJoin`.

abos

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

locations

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

We want to make a new RDD with only these!

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

Outer Joins (`leftOuterJoin`, `rightOuterJoin`)

We can do the converse using a `rightOuterJoin`.

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

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

Outer Joins (leftOuterJoin, rightOuterJoin)

Example continued with concrete data:

customersWithLocationDataAndOptionalAbos

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

customer# Option[(lastName,kindOfAbo)] frequentCity

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

Outer Joins (leftOuterJoin, rightOuterJoin)

Example continued with concrete data:

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

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

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

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

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

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

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

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

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

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

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

Shuffles Happen

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

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