Big Data Analytics with Apache Spark (Part 2)

Nastaran Fatemi

What we've seen so far

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

In the following

1. We'll discover Pair RDDs (key-value pairs)
2. We'll learn about Pair RDD specific operators
3. We'll get a glimpse of what "shuffling" is, and why it hits performance (latency)

Pair RDDs (Key-Value Pairs)

Often when working with distributed data, it’s useful to organize data into **key-value pairs**. In Spark, these are Pair RDDs.

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

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

Depending on the operation, data in an RDD may have to be **shuffled** among worker nodes, using worker-worker communication.

This is often the case for many operations of Pair RDDs!

Pair RDDs

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.

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 lines = sc.textFile("README.md")
val pairs = lines.map(x=>(x.split(“ “)(0), x))
```

In Scala, for the functions on keyed data to be available, we need to return tuples. An implicit conversion on RDDs of tuples exists to provide the additional key/value functions.

Pair RDD Transformation: groupByKey

`groupByKey` groups values which have the same key.

When called on a dataset of (K, V) pairs, returns a dataset of (K, Iterable<V>) pairs.

```
val data = Array((1, 2), (3, 4), (3, 6))
val myRdd= sc.parallelize(data)
val groupedRdd = myRdd.groupByKey()
groupedRdd.collect.foreach(println)
   (1,CompactBuffer(2))
   (3,CompactBuffer(4, 6))
```

Note: *If you are grouping in order to perform an aggregation (such as a sum or average) over each key, using `reduceByKey` or `aggregateByKey` will yield much better performance.*

groupByKey
another exmaple

```
case class Event(organizer: String, name: String, budget: Int)

val events : List [Event] = List (
        Event("HEIG-VD", "Baleinev", 42000),
        Event("HEIG-VD","Stages WINS",14000),
        Event("HEIG-VD","CyberSec",20000),
        Event("HE-ARC","Portes ouvertes",25000),
        Event("HE-ARC","Cérmonie diplômes",10000),
        ...
        )

val eventsRdd = sc.parallelize(events)
val eventsKvRdd = eventsRdd.map (event => (event.organizer,
                                           event.budget))
val groupedRdd = eventsKvRdd.groupByKey()
groupedRdd.collect().foreach(println)

//(HEIG-VD,CompactBuffer(42000, 14000, 20000))
//(HE-ARC,CompactBuffer(25000, 10000))
// ...
```

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)]
```

When called on a dataset of (K, V) pairs, returns a dataset of (K, V) pairs where the values for each key are aggregated using the given reduce function *func*, which must be of type (V,V) => V.

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

```
val eventsRdd = sc.parallelize(events)
val eventsKvRdd = eventsRdd.map (event => (event.organizer,
                                           event.budget))
val totalBudget = eventsKvRdd.reduceByKey(_+_)
groupedRdd.collect().foreach(println)
//(HEIG-VD,76000)
//(HE-ARC,35000)
//...
```

Pair RDD Transformation: mapValues and Action: countByKey

```
def mapValues[U](f: (V) ⇒ U): RDD[(K, U)]
```

mapValues 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 value) containing (budget, #events)
val intermediate =
  eventsKvRdd.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 =
  eventsKvRdd.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)
//(HEIG-VD,25333)
//(HE-ARC,17500)
```

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 difference between the two types of joins is exactly the same as in databases

Inner Joins (join)

```
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)

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

Example: Let's pretend the CFF has two datasets. One RDD representing Customers, their names 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"))
val locations = sc.parallelize(ls)

val trackedCustomers = abos.join(locations)

trackedCustomers.foreach(println)
// 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))
```

Optimizing...

Now that we understand Spark's programming model, and a majority of Spark's key operations, we'll now see how we can optimize what we do with Spark to keep it practical.

It's very easy to write code that takes tens of minutes to compute when it could be computed in only tens of seconds!

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 CFF mobile app have made in the past month.

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(...)
```

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
```

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]]
```

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]]
              .map(p => (p._1, (p._2.size, p._2.sum)))
              .collect()
```

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?

Grouping and Reducing, Example: What's Happening?

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

Starting with purchasesRdd:

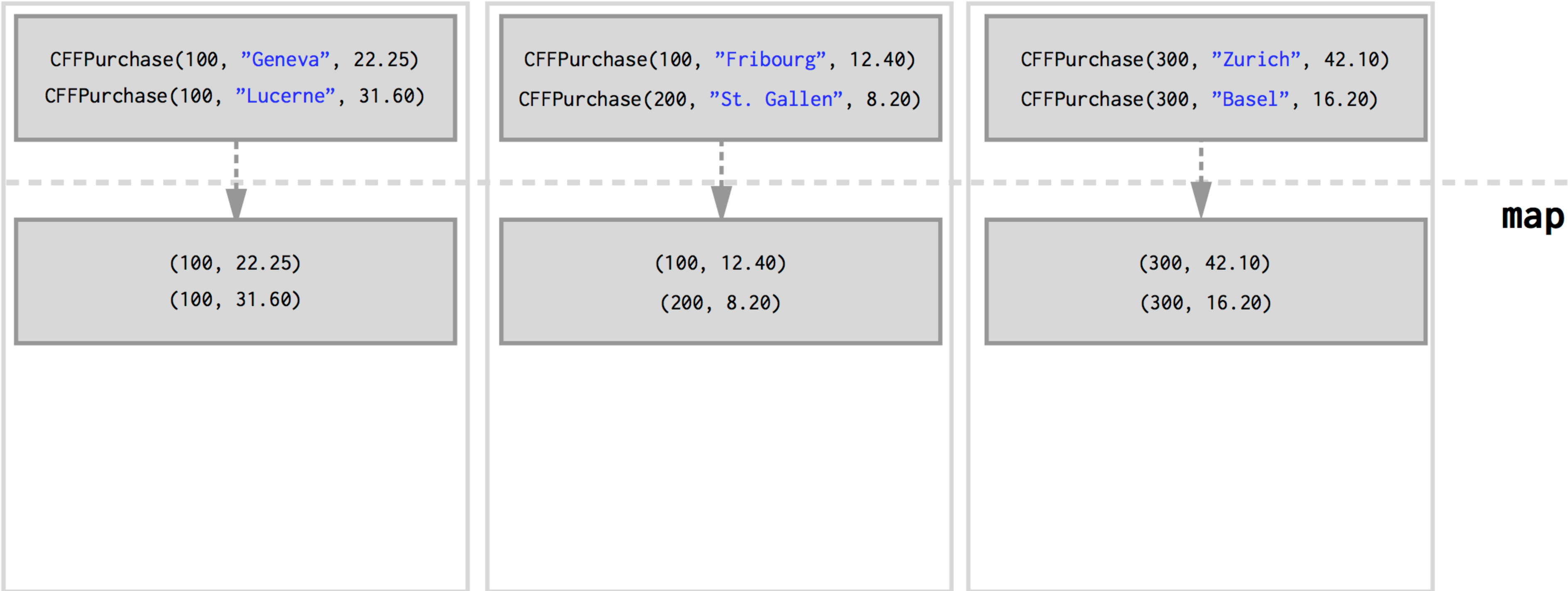
```
CFFPurchase(100, "Geneva", 22.25)
CFFPurchase(100, "Lucerne", 31.60)
```

```
CFFPurchase(100, "Fribourg", 12.40)
CFFPurchase(200, "St. Gallen", 8.20)
```

```
CFFPurchase(300, "Zurich", 42.10)
CFFPurchase(300, "Basel", 16.20)
```

Grouping and Reducing, Example: What's Happening?

What might this look like on the cluster?



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.

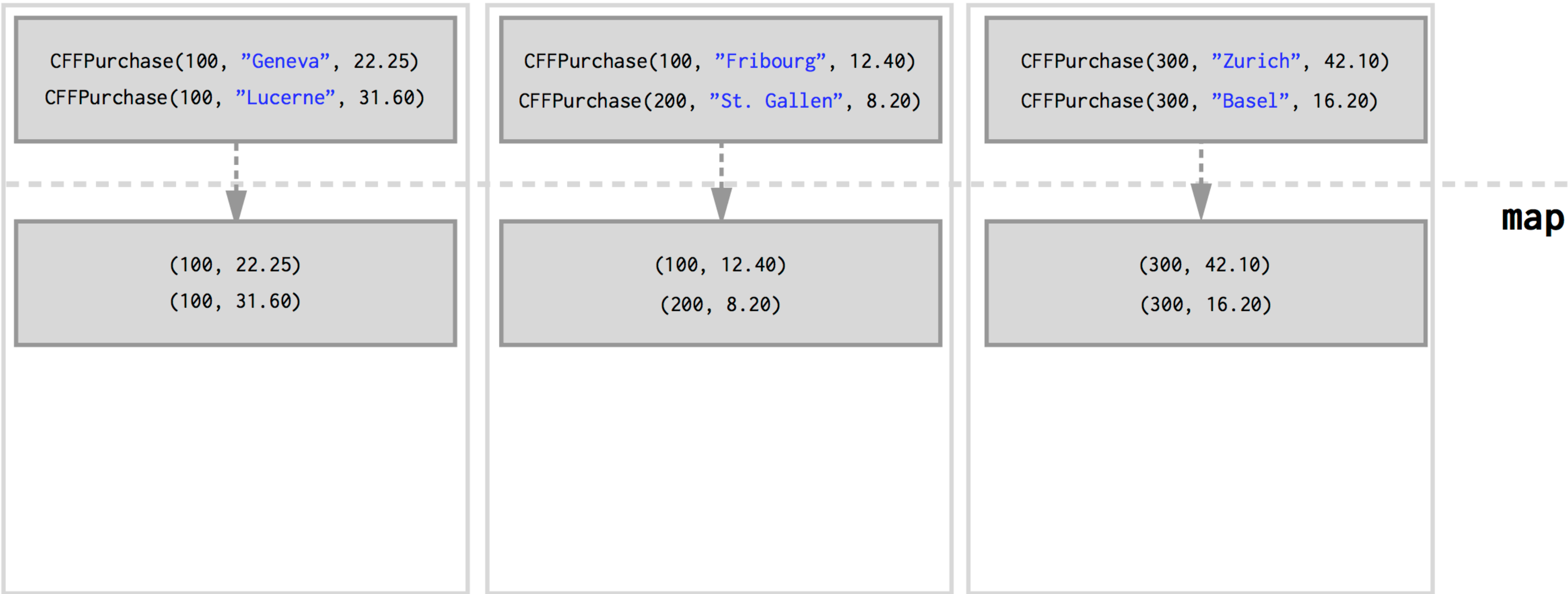
```
val purchasesRdd: RDD[CFFPurchase] = sc.textFile(...)

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.

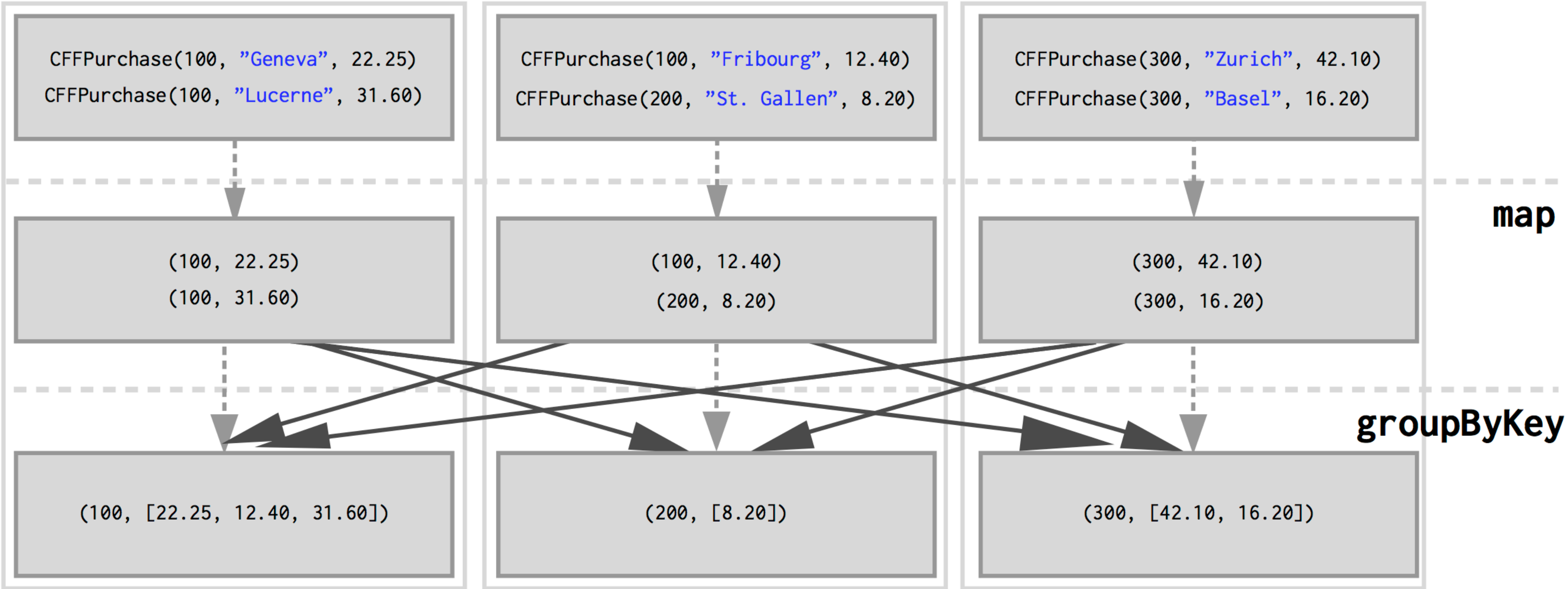
Grouping and Reducing, Example: What's Happening?

What might this look like on the cluster?



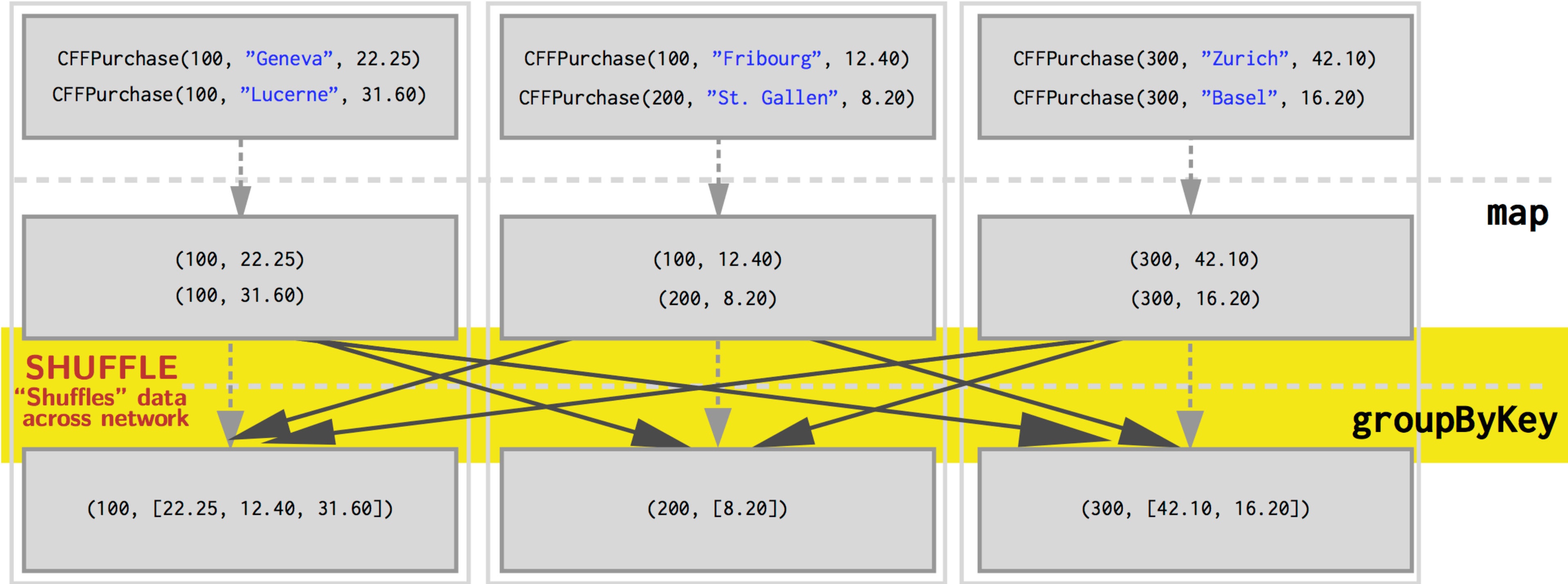
Grouping and Reducing, Example: What's Happening?

What might this look like on the cluster?



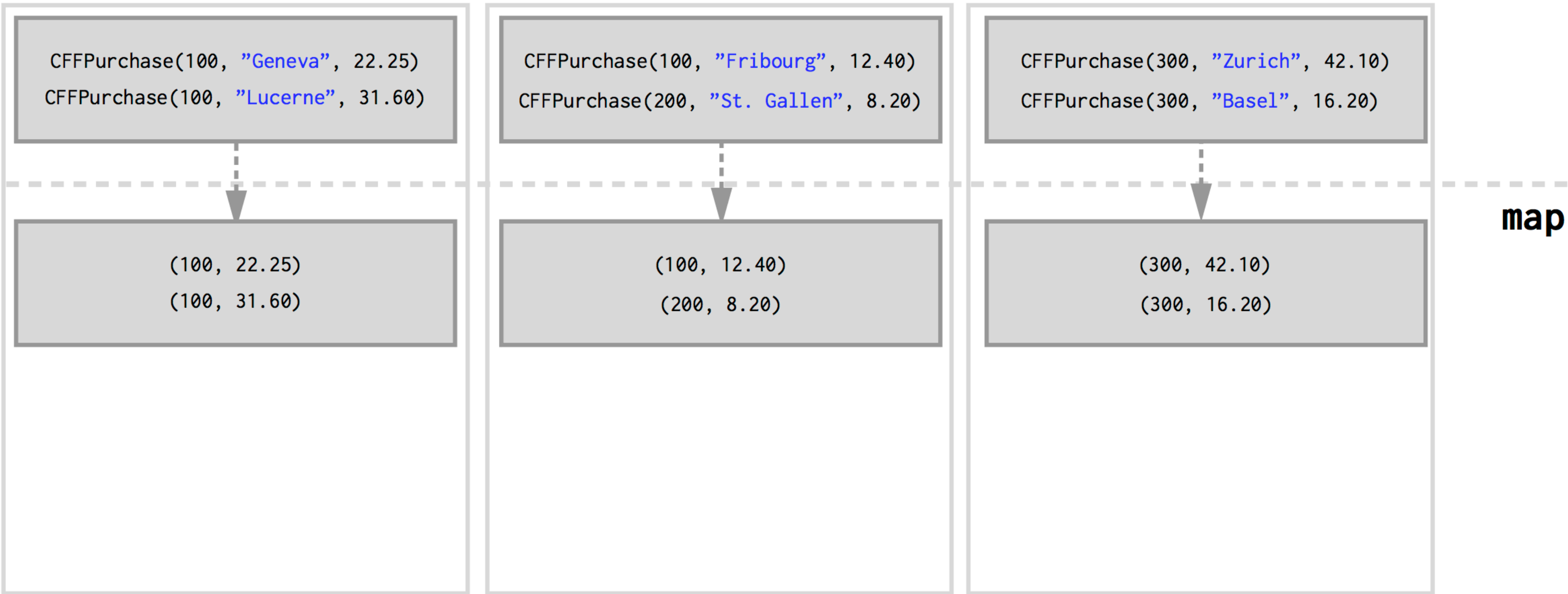
Grouping and Reducing, Example: What's Happening?

What might this look like on the cluster?



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.

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)]
```

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?

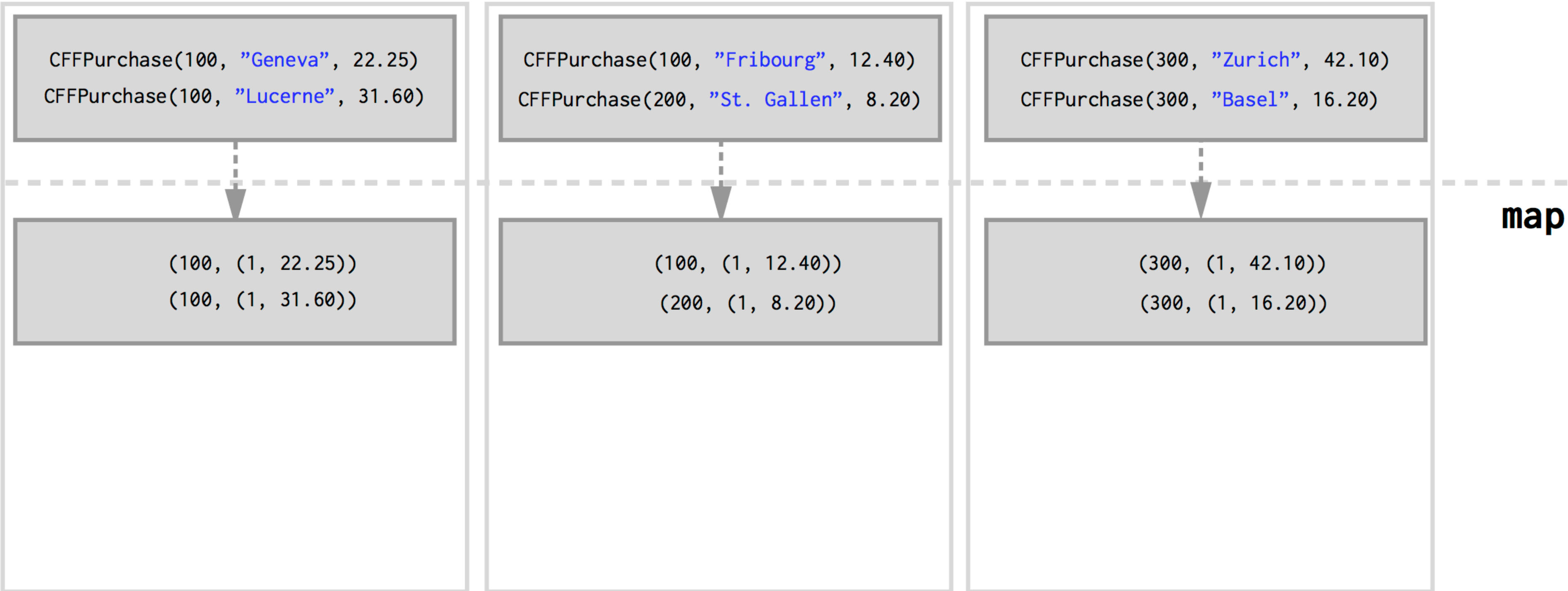
Grouping and Reducing, Example Optimized

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

What might this look like on the cluster?

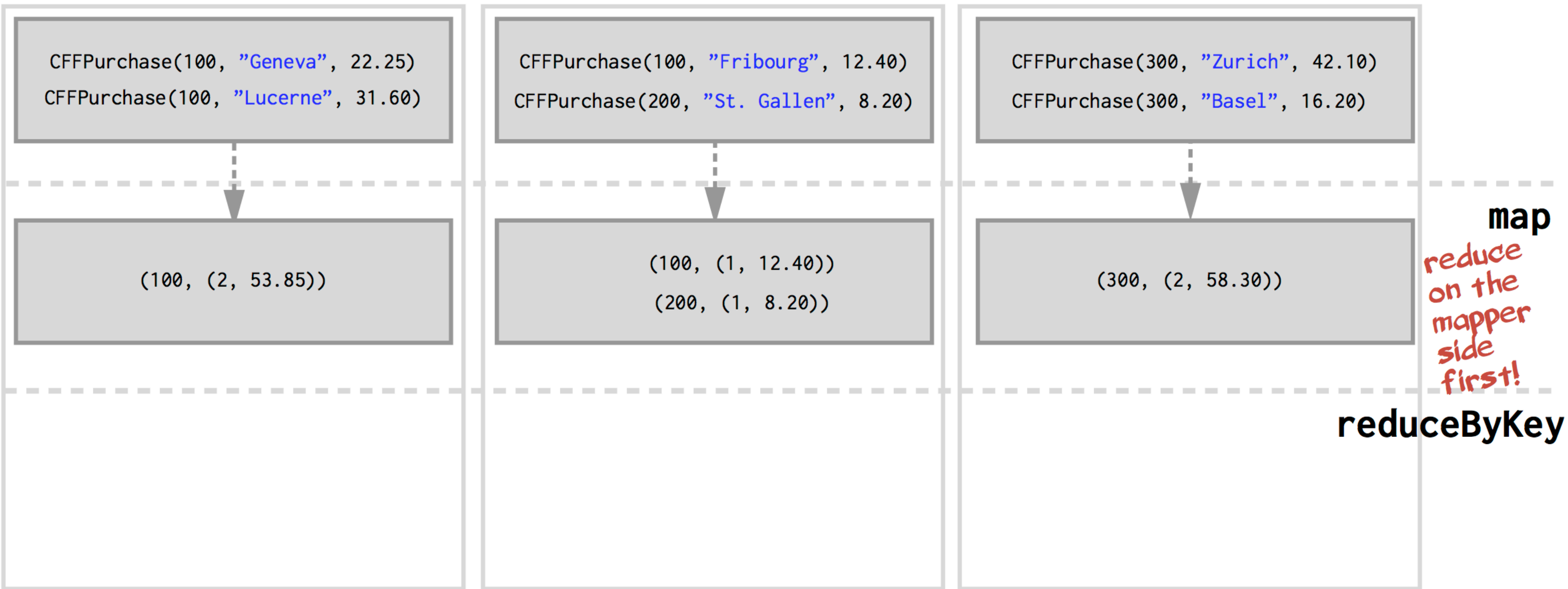
Grouping and Reducing, Example Optimized

What might this look like on the cluster?



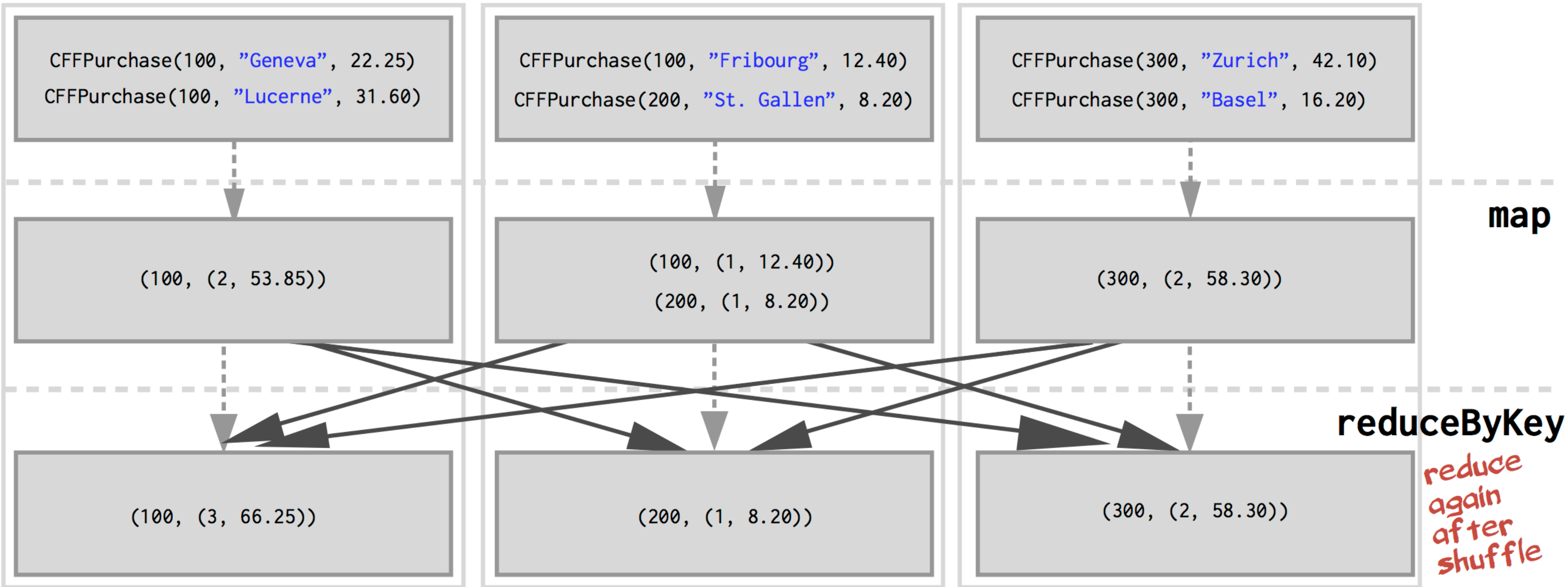
Grouping and Reducing, Example Optimized

What might this look like on the cluster?



Grouping and Reducing, Example Optimized

What might this look like on the cluster?



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!

groupByKey and reduceByKey Running Times

Benchmark results on a real cluster:

```
> 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
```

References

The content of this section is partly taken from the slides of the course "Parallel Programming and Data Analysis" by Heather Miller at EPFL.

Other references used here:

- *Learning Spark*
  by Holden Karau, Andy Konwinski, Patrick Wendell & Matei Zaharia. O'Reilly, February 2015.
- Spark documentation, available at
  http://spark.apache.org/docs/latest/