Shuffling: What it is and why it's important

Big Data Analysis with Scala and Spark

Heather Miller

```
?? 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?**

?? 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
```

?? 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".

?? 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!**

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.

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 = ...
```

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(...)

// 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()
```

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`:

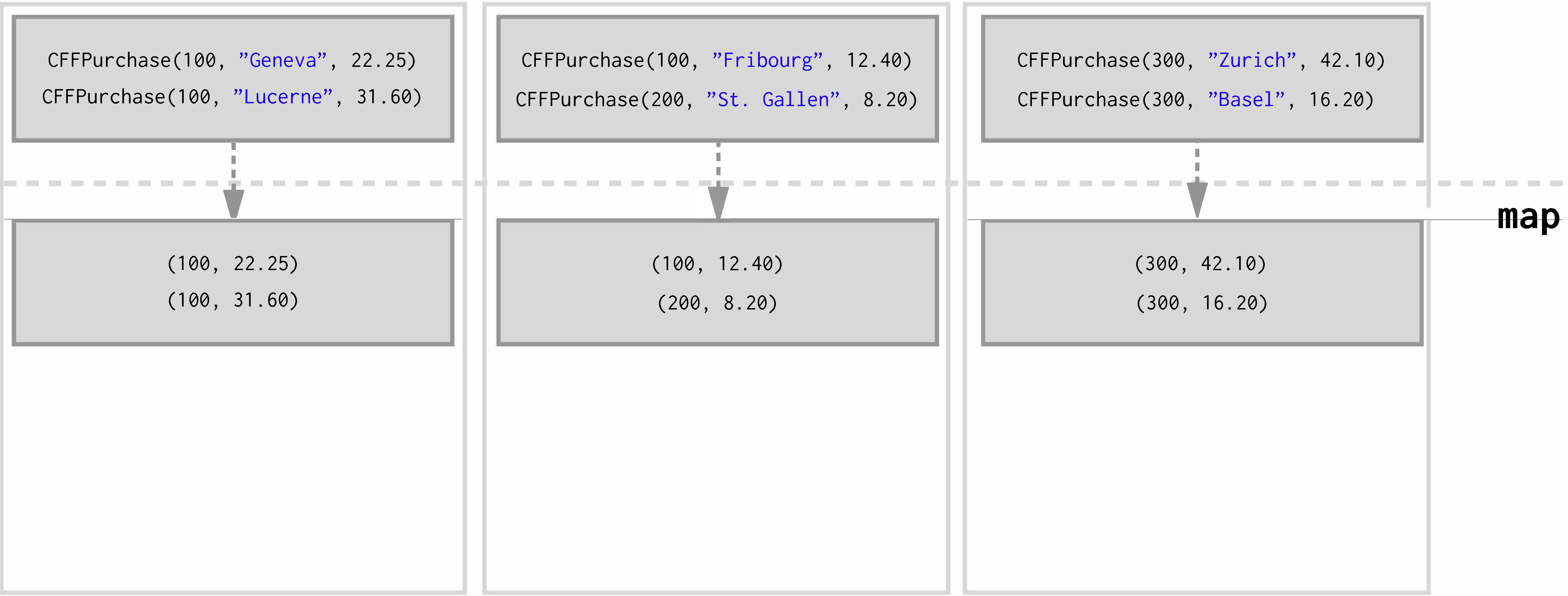
```
CFFPurchase(100, "Geneva", 22.25)
CFFPurchase(100, "Lucerne", 31.60)
```

```
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]]
```

Grouping and Reducing, Example

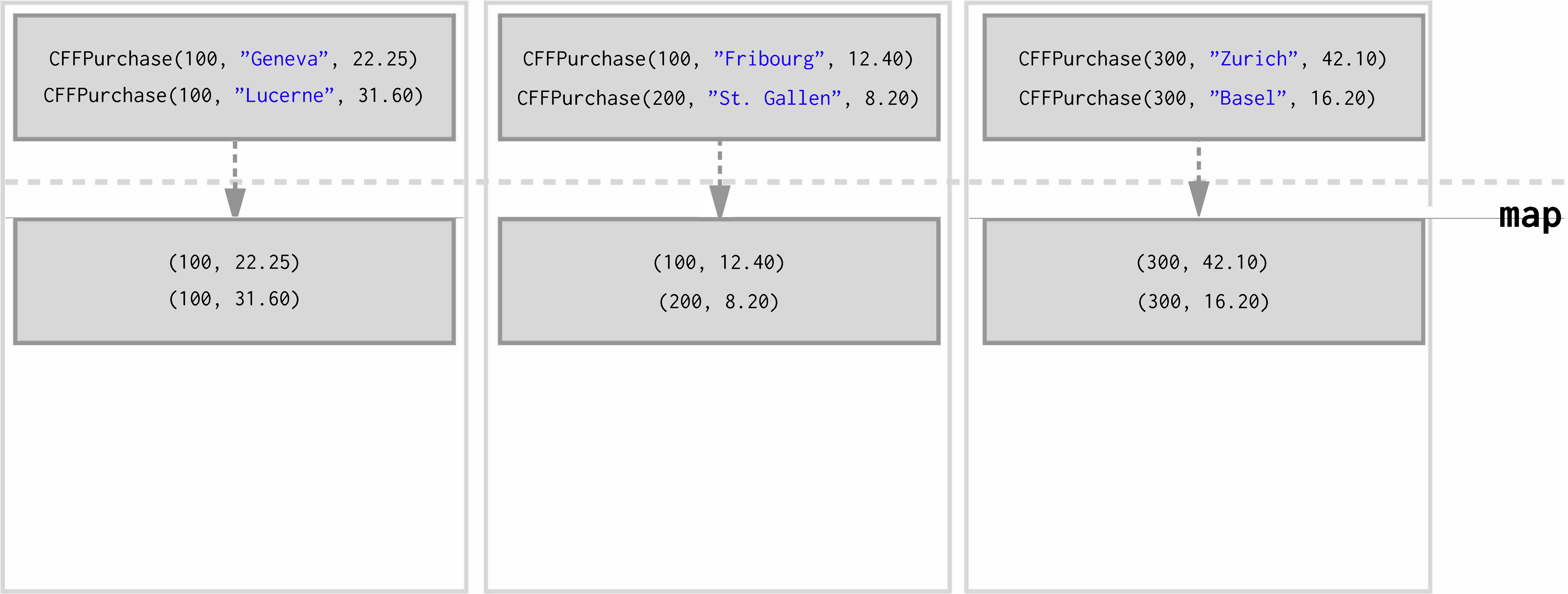
Goal: calculate how many trips, and how much money was spent by each individual customer over the course of the month.

```
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.

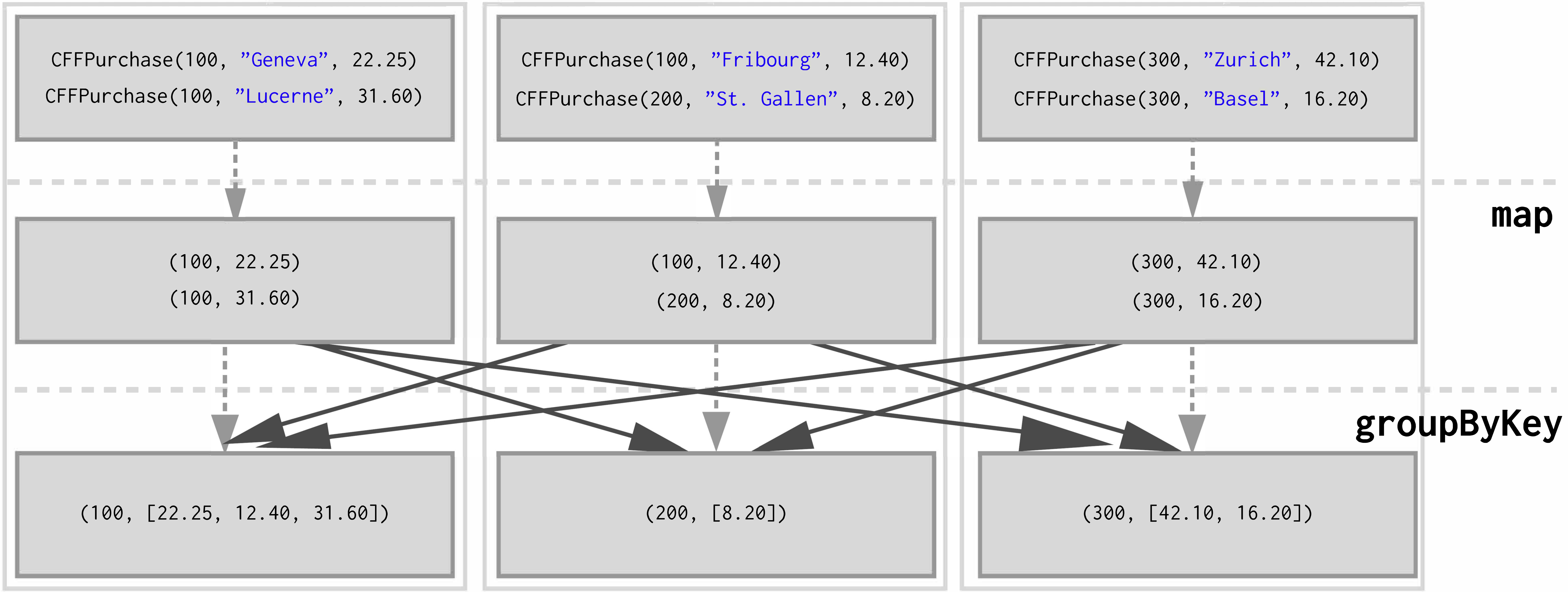
Grouping and Reducing, Example – What's Happening?

What might this look like on the cluster?



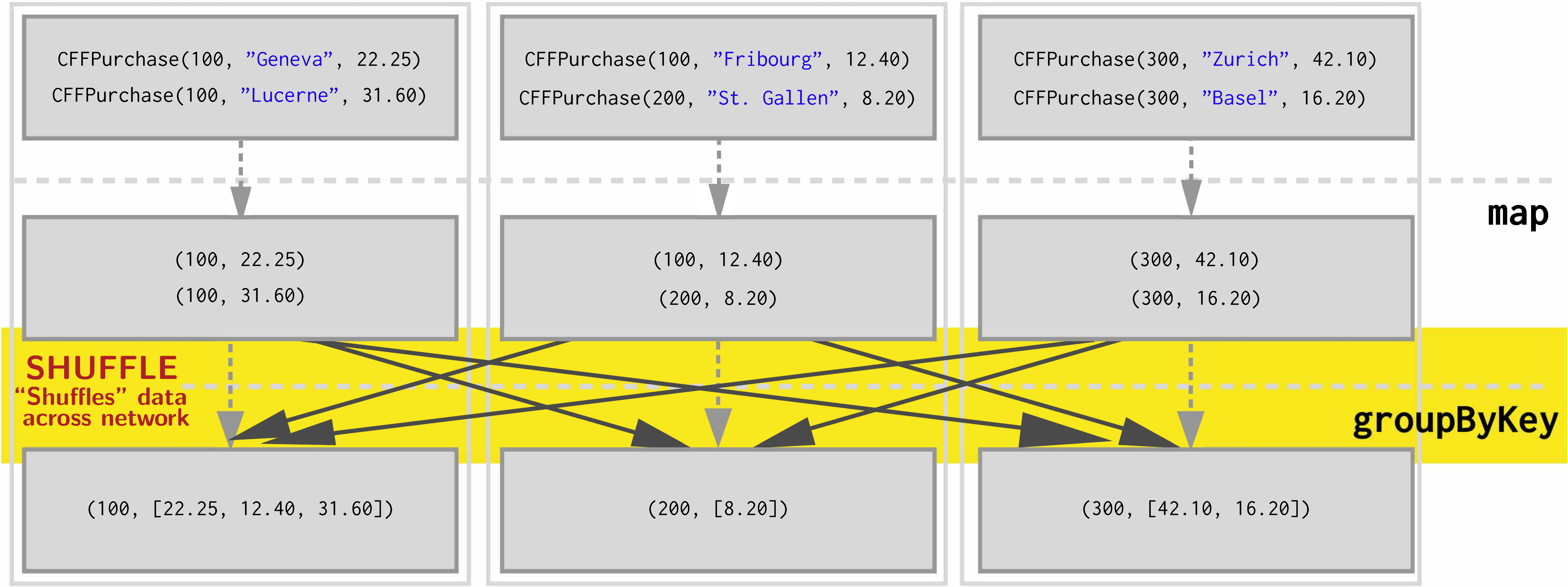
Grouping and Reducing, Example – What's Happening?

What might this look like on the cluster?



Grouping and Reducing, Example – What's Happening?

What might this look like on the cluster?



Reminder: Latency Matters (Humanized)

Shared Memory

Seconds

```
L1 cache reference..........0.5s
L2 cache reference............7s
Mutex lock/unlock...........25s
```

Minutes

```
Main memory reference.....1m 40s
```

Distributed

Days

```
Roundtrip within
same datacenter.........5.8 days
```

Years

```
Send packet
CA->Netherlands->CA....4.8 years
```

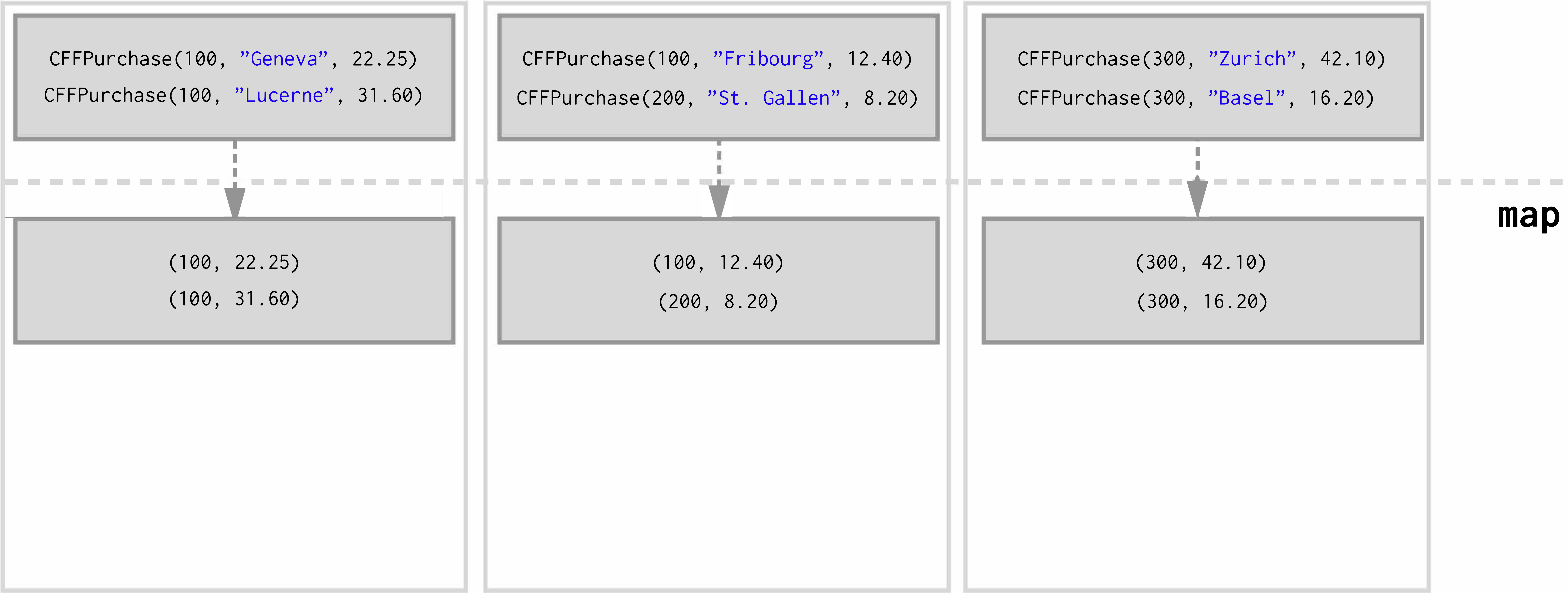
We don't want to be sending all of our data over the network if it's not absolutely required. Too much network communication kills performance.

Can we do a better job?

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

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(...) // ?
```

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(...) // ?
```

(Int, Double)

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

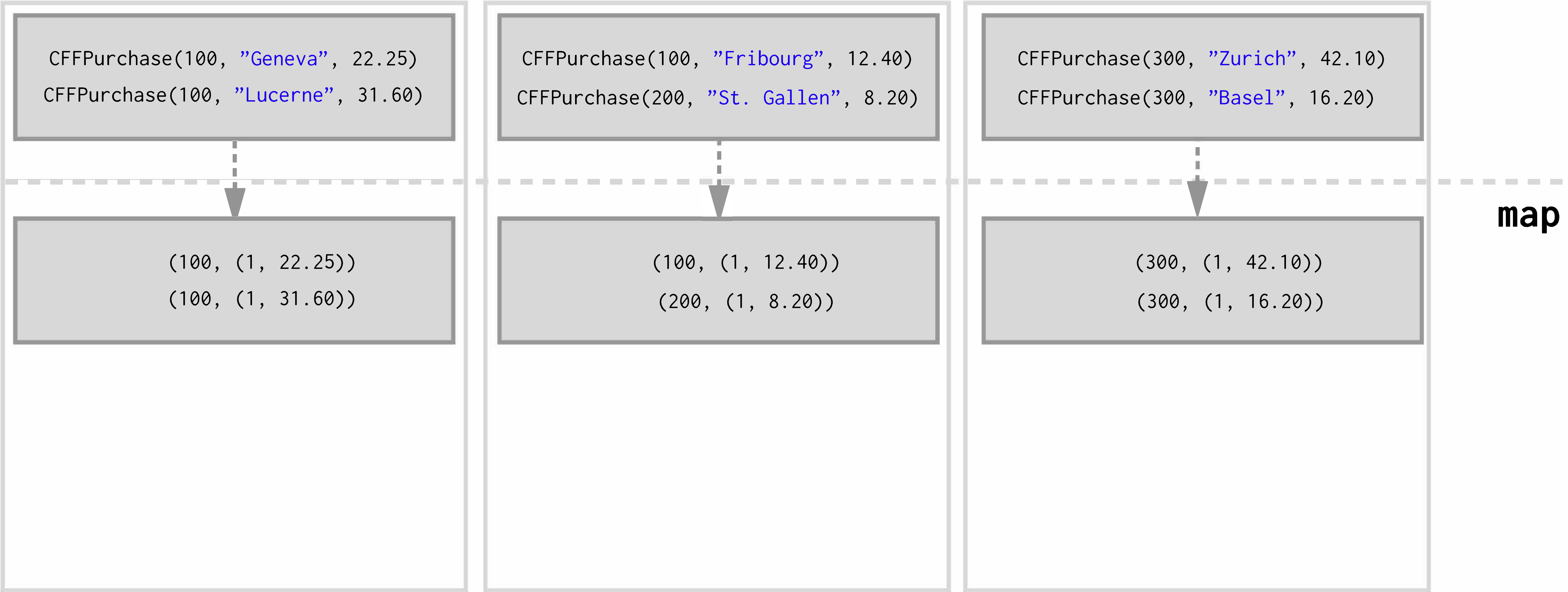
Grouping and Reducing, Example – Optimized

```
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?

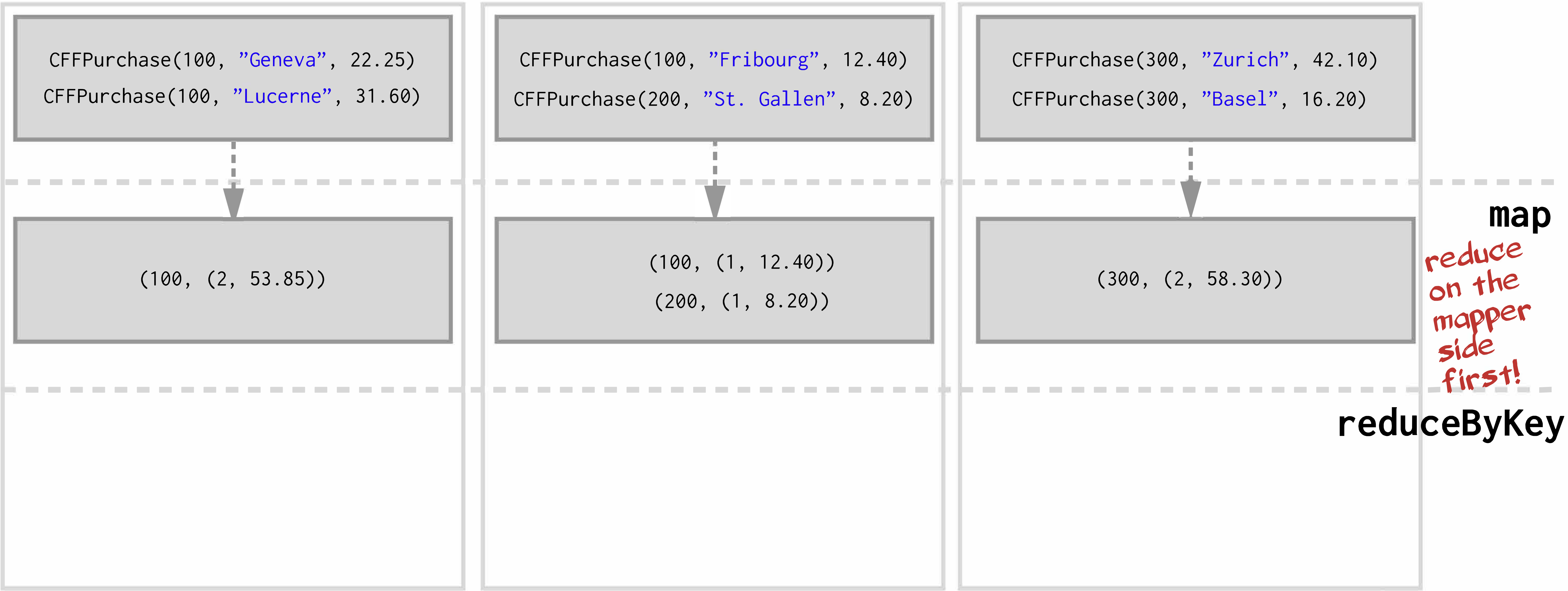
Grouping and Reducing, Example – Optimized

What might this look like on the cluster?



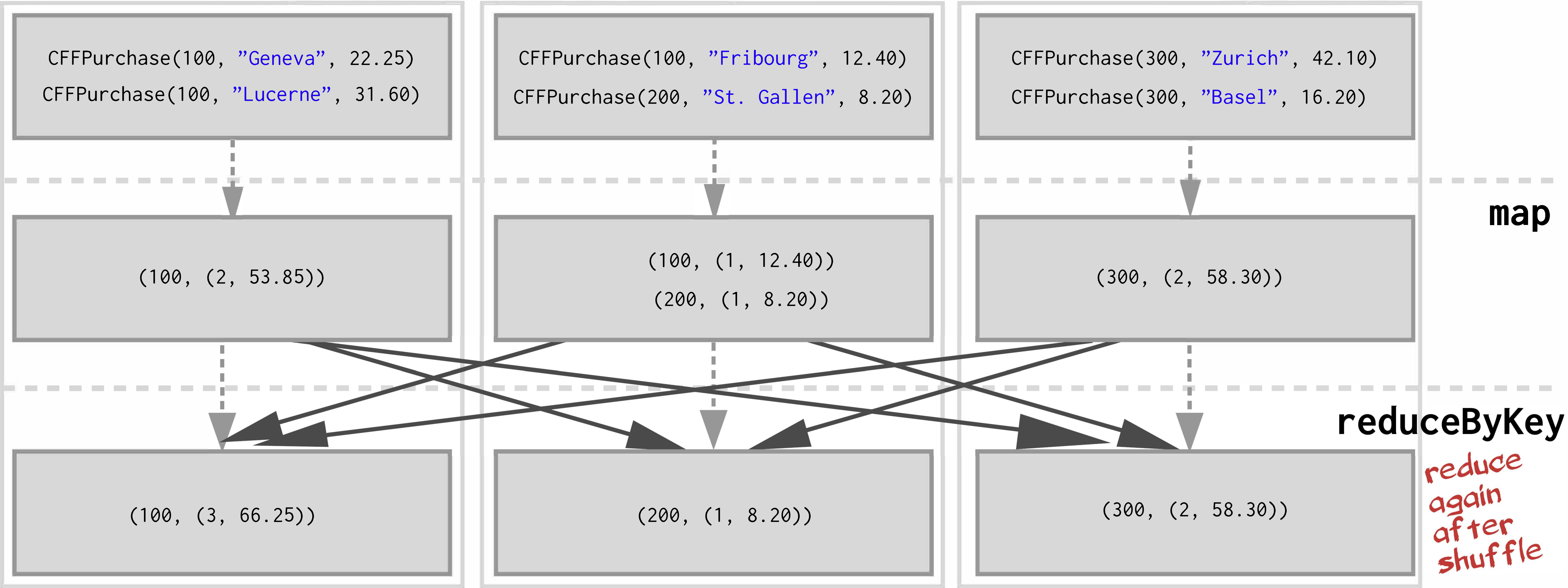
Grouping and Reducing, Example – Optimized

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?

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!

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.

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
```

Shuffling

Recall our example using groupByKey:

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

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?

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.