



ÉCOLE POLYTECHNIQUE  
FÉDÉRALE DE LAUSANNE

# Shuffling, Partitioning, and Closures

Principles of Functional Programming

Heather Miller

## What we've learned so far

- ▶ We extended *data parallel programming* to the distributed case.
- ▶ We saw that Apache Spark implements this distributed data parallel model.
- ▶ We developed some intuition for how long it takes to do various operations – in memory, on disk, and over the network. And, specifically, we saw how important it is to reduce network communication.

# What we've learned so far

## Spark's Basic Programming Model

- ▶ We saw that, at a glance, Spark looks like Scala collections
- ▶ However, Spark behaves **very 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 learned in detail about reduction operations in Spark vs Scala collections

## What we've learned so far

### **Distributed Key-Value Pairs** (Pair RDDs)

- ▶ We got a sampling of Spark's key-value pairs (Pair RDDs)
- ▶ We saw all of the different sorts of joins
- ▶ We learned other important operations on just Pair RDDs
- ▶ We got a glimpse of “shuffling”

# Today...

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 clear code that takes tens of minutes to compute when it could be computed in only tens of seconds.*

## 1. **Shuffling**

- ▶ What is it and why is it important?
- ▶ How do I know when it happens?
- ▶ How can I optimize an operation that requires a shuffle?

## 2. **Partitioning**

## 3. **Closures and Capturing**

## 4. **Shared Variables**

## 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.

```
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:

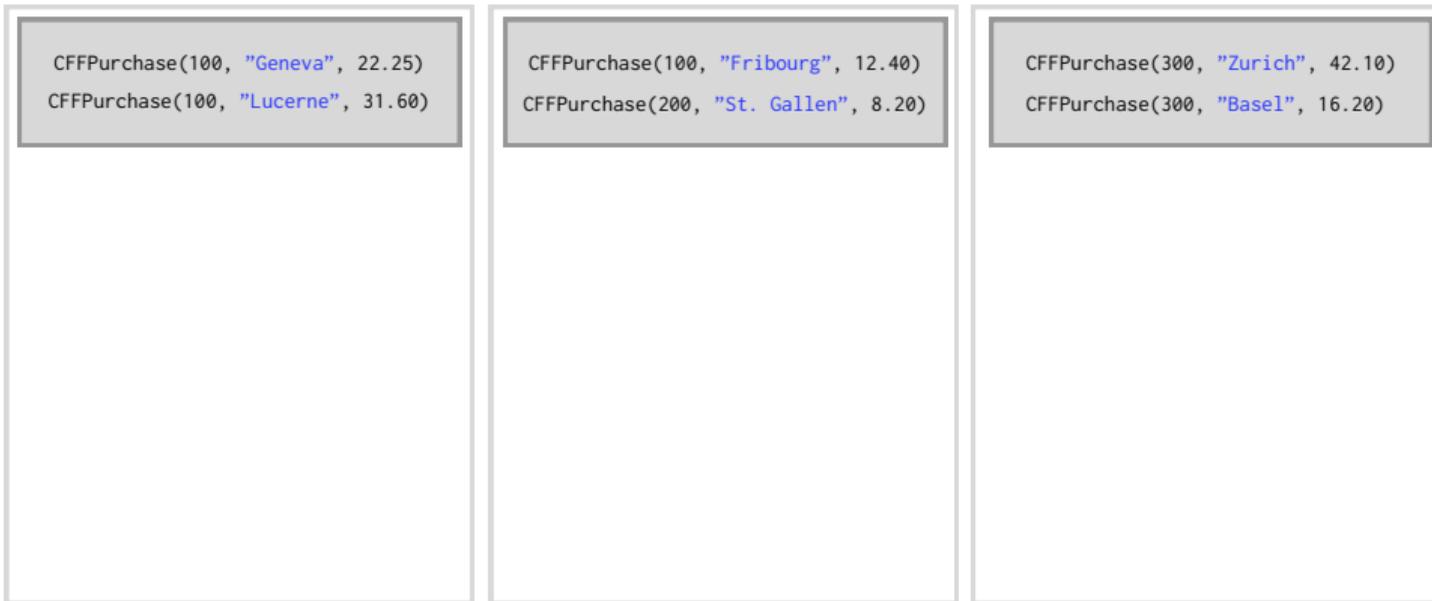
```
val purchases = List(CFFPurchase(100, "Geneva", 22.25),  
                    CFFPurchase(300, "Zurich", 42.10),  
                    CFFPurchase(100, "Fribourg", 12.40),  
                    CFFPurchase(200, "St. Gallen", 8.20),  
                    CFFPurchase(100, "Lucerne", 31.60),  
    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



# 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.

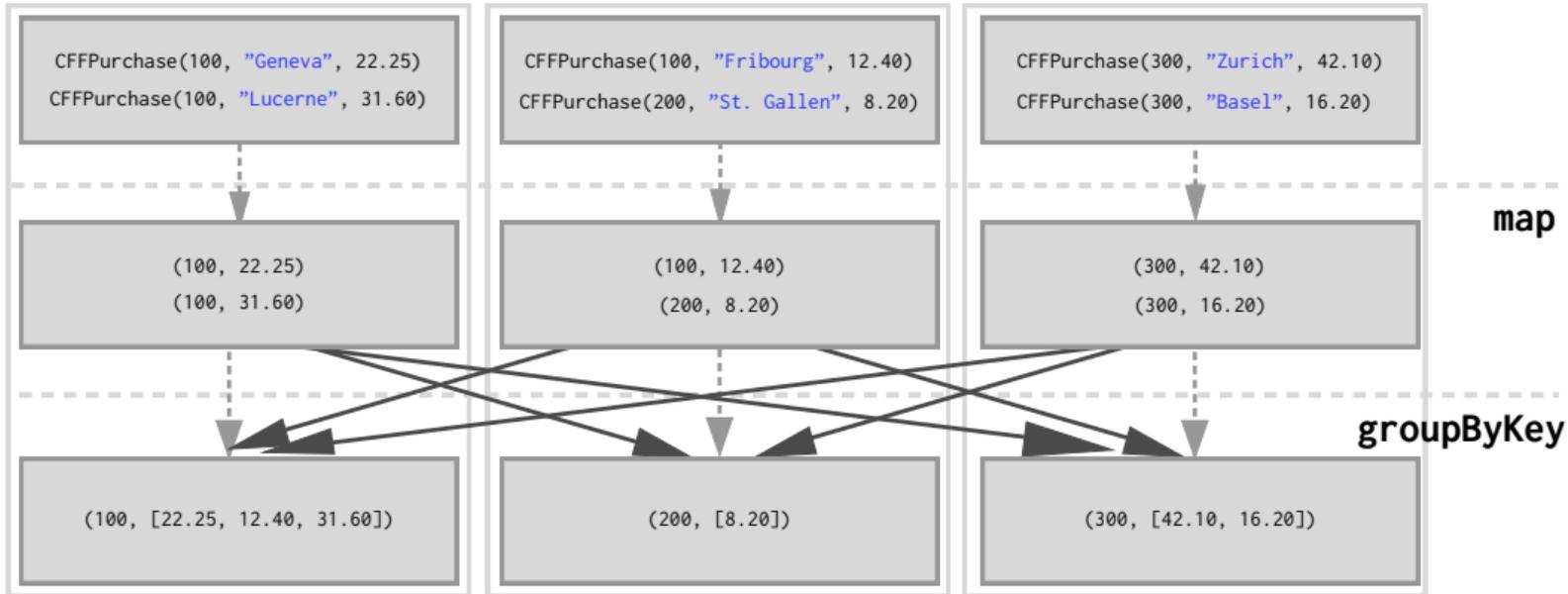
# Grouping and Reducing, Example – What's Happening?

What might this look like on the cluster?



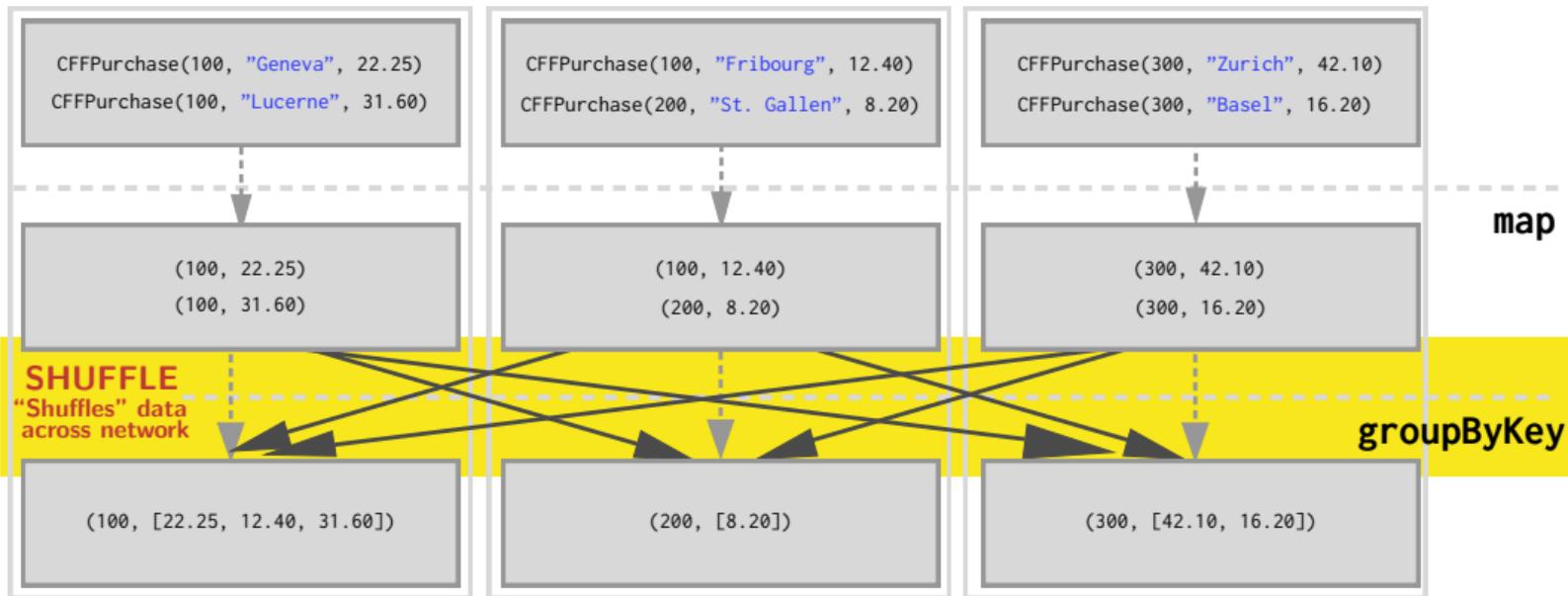
# Grouping and Reducing, Example – What's Happening?

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(...) // ?
```

## Grouping and Reducing, Example – Optimized

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

Recall that we're reducing over the *values per key*.

Since our values are an `Iterable[(Int, Double)]`, the function that we pass to `reduceByKey` must reduce over two such pairs.

## 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()
```

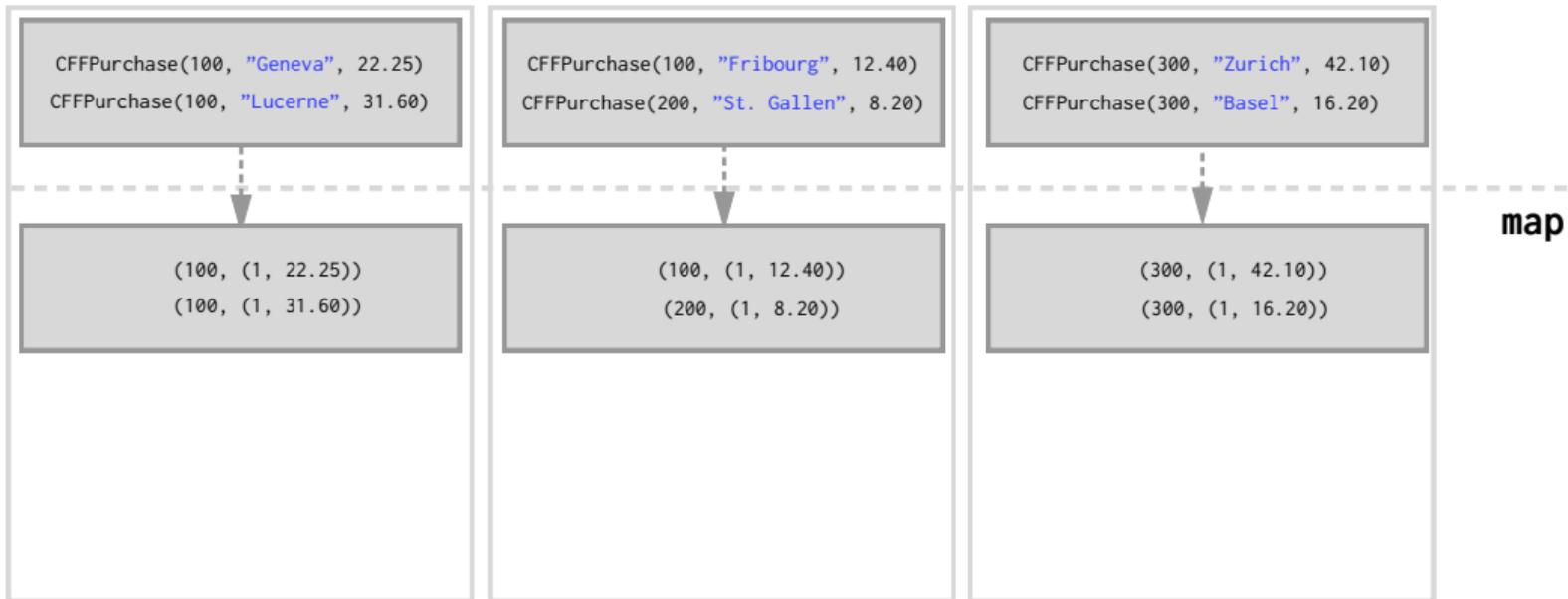
## Grouping and Reducing, Example – Optimized

```
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?**

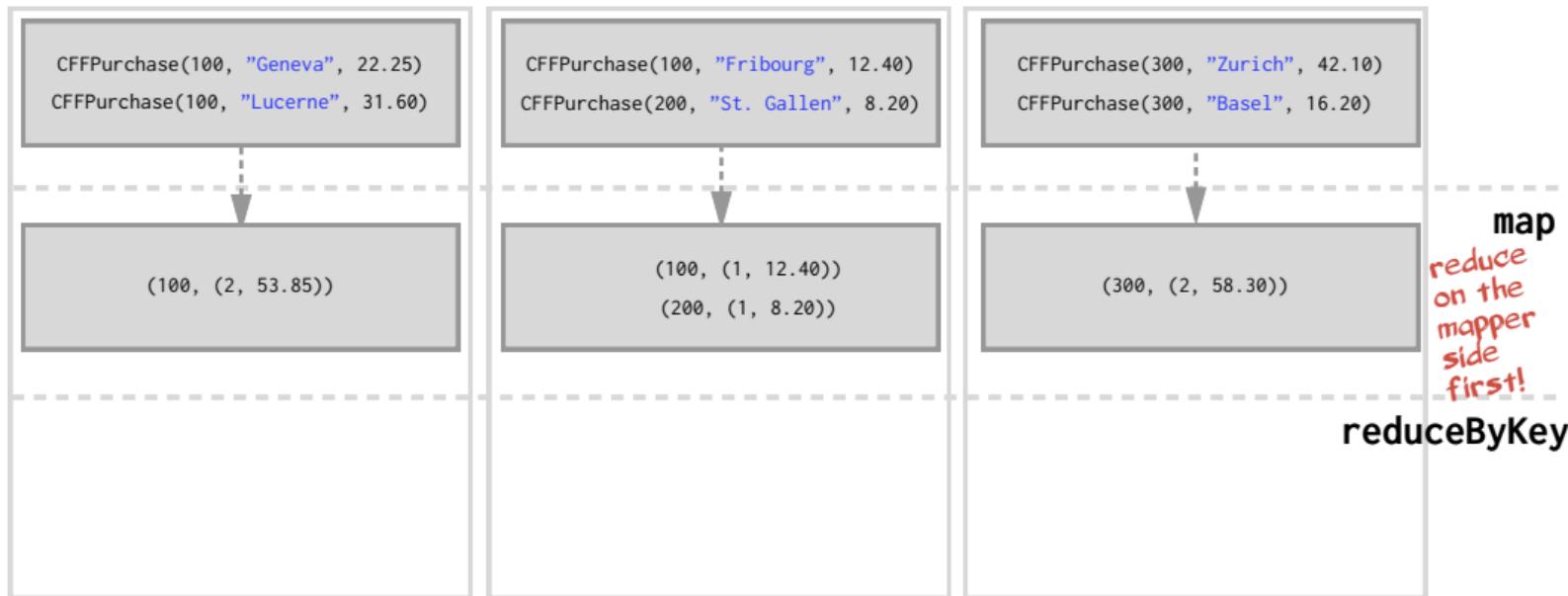
# Grouping and Reducing, Example – Optimized

What might this look like on the cluster?



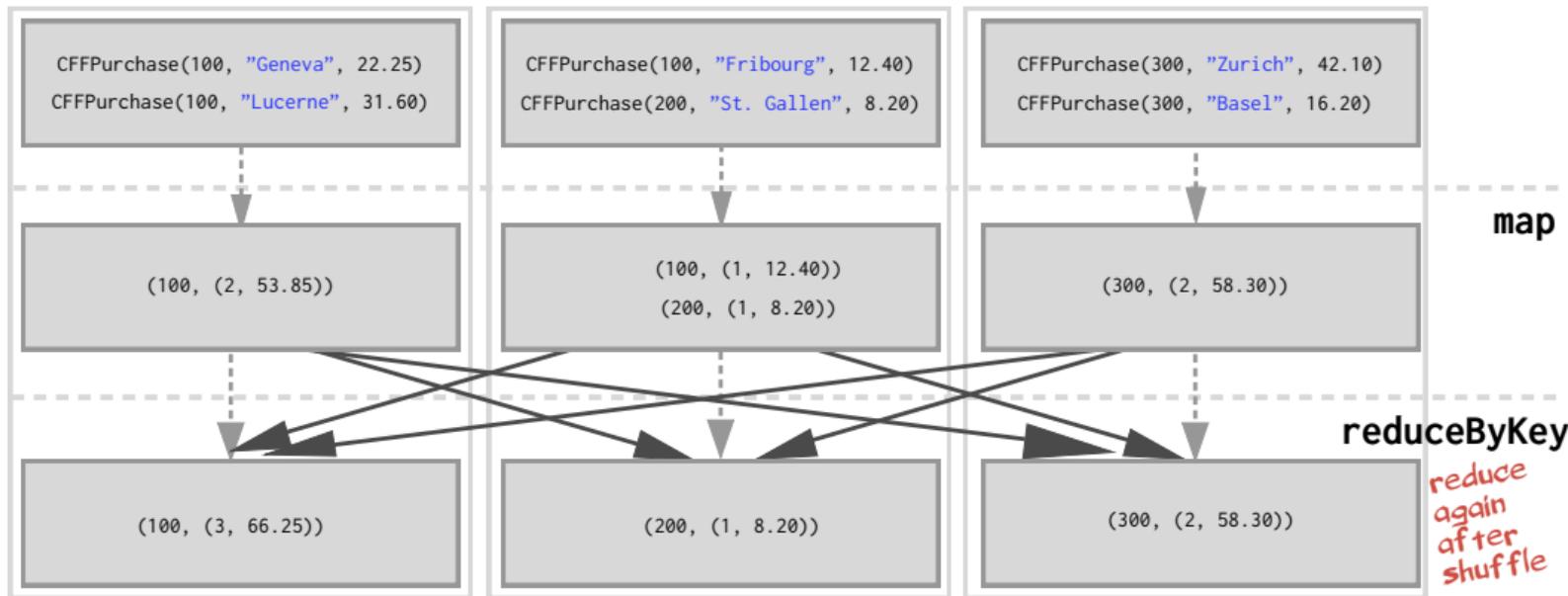
# Grouping and Reducing, Example – Optimized

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-trivial 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-trivial 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

*Full example with 20 million element RDD can be found in the notebook on our Databricks Cloud installation.*

# 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.

# “Partitioning”?

**First, a quick detour into partitioning...**

# Partitions

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

## **Properties of partitions:**

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

## **Two kinds of partitioning available in Spark:**

- ▶ Hash partitioning
- ▶ Range partitioning

**Customizing a partitioning is only possible on Pair RDDs.**

## Hash partitioning

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

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

## Hash partitioning

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

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

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

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

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

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

## Range partitioning

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

- ▶ Examples: Int, Char, String, ...

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

Using a range partitioner, keys are partitioned according to:

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

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

## Hash Partitioning: Example

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

## Hash Partitioning: Example

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

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

## Hash Partitioning: Example

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

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

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

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

The result is a very unbalanced distribution which hurts performance.

## Range Partitioning: Example

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

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

## Range Partitioning: Example

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

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

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

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

The resulting partitioning is much more balanced.

# Partitioning Data

**How do we set a partitioning for our data?**

# Partitioning Data

## How do we set a partitioning for our data?

There are two ways to create RDDs with specific partitionings:

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

## Partitioning Data: `partitionBy`

Invoking `partitionBy` creates an RDD with a specified partitioner.

## Partitioning Data: partitionBy

Invoking `partitionBy` creates an RDD with a specified partitioner.

Example:

```
val pairs = purchasesRdd.map(p => (p.customerId, p.price))
```

## Partitioning Data: partitionBy

Invoking `partitionBy` creates an RDD with a specified partitioner.

Example:

```
val pairs = purchasesRdd.map(p => (p.customerId, p.price))
```

```
val tunedPartitioner = new RangePartitioner(8, pairs)
```

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

## Partitioning Data: `partitionBy`

Invoking `partitionBy` creates an RDD with a specified partitioner.

Example:

```
val pairs = purchasesRdd.map(p => (p.customerId, p.price))
```

```
val tunedPartitioner = new RangePartitioner(8, pairs)
```

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

Creating a `RangePartitioner` requires:

1. Specifying the desired number of partitions.
2. Providing a Pair RDD with *ordered keys*. This RDD is *sampled* to create a suitable set of *sorted ranges*.

## Partitioning Data: `partitionBy`

Invoking `partitionBy` creates an RDD with a specified partitioner.

Example:

```
val pairs = purchasesRdd.map(p => (p.customerId, p.price))
```

```
val tunedPartitioner = new RangePartitioner(8, pairs)
```

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

Creating a `RangePartitioner` requires:

1. Specifying the desired number of partitions.
2. Providing a Pair RDD with *ordered keys*. This RDD is *sampled* to create a suitable set of *sorted ranges*.

**Important:** the result of `partitionBy` should be persisted. Otherwise, the partitioning is repeatedly applied (involving shuffling!) each time the partitioned RDD is used.

# Partitioning Data Using Transformations

## **Partitioner from parent RDD:**

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

## **Automatically-set partitioners:**

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

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

## Partitioning Data Using Transformations

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

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

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

## Partitioning Data Using Transformations

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

**Why?**

## Partitioning Data Using Transformations

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

### **Why?**

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

## Partitioning Data Using Transformations

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

### **Why?**

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

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

## Partitioning Data Using Transformations

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

### Why?

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

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

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

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

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

## Optimization using range partitioning

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

## Optimization using range partitioning

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

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

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

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

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

# Optimization using range partitioning

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

purchasesPerMonthSlowLarge: Long = 100000

Command took 15.48s

---

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

purchasesPerMonthFastLarge: Long = 100000

Command took 4.65s

---

## On the range partitioned data:

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

purchasesPerMonthFasterLarge: Long = 100000

Command took 1.79s

# Optimization using range partitioning

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

purchasesPerMonthSlowLarge: Long = 100000

Command took 15.48s

---

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

purchasesPerMonthFastLarge: Long = 100000

Command took 4.65s

## On the range partitioned data:

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

purchasesPerMonthFasterLarge: Long = 100000

Command took 1.79s

almost a 9x speedup over  
purchasePerMonthSlowLarge!

## Partitioning Data: `partitionBy`, Another Example

*From pages 61-64 of the Learning Spark book*

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

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

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

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

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

## Partitioning Data: partitionBy, Another Example

*From pages 61-64 of the Learning Spark book*

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

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

**Is this OK?**

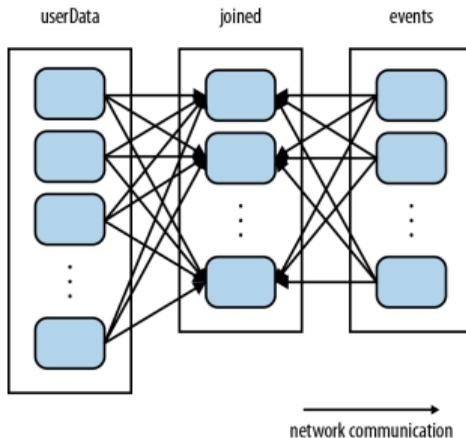
## Partitioning Data: partitionBy, Another Example

*From pages 61-64 of the Learning Spark book*

**It will be very inefficient!**

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

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



## Partitioning Data: partitionBy, Another Example

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

## Partitioning Data: `partitionBy`, Another Example

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

Therefore, `userData` becomes:

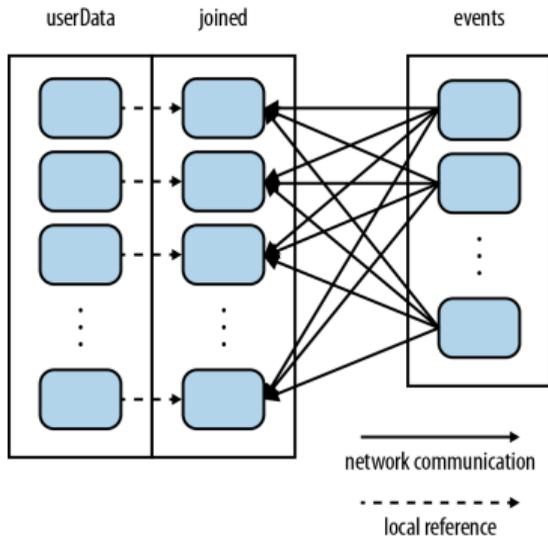
```
val userData = sc.sequenceFile[UserID, UserInfo]("hdfs://...")
    .partitionBy(new HashPartitioner(100)) // Create 100 partitions
    .persist()
```

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

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

# Partitioning Data: `partitionBy`, Another Example

Or, shown visually:



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

## Back to shuffling

Recall our example using `groupByKey`:

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

## Back to shuffling

Recall our example using `groupByKey`:

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

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

## Back to shuffling

Recall our example using `groupByKey`:

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

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

Grouping is done using a hash partitioner with default parameters.

## Back to shuffling

Recall our example using `groupByKey`:

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

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

Grouping is done using a hash partitioner with default parameters.

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

## How do I know a shuffle will occur?

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

## How do I know a shuffle will occur?

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

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

## How do I know a shuffle will occur?

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

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

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

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

```
partitioned.reduceByKey((v1, v2) => (v1._1 + v2._1, v1._2 + v2._2))  
    .toDebugString
```

```
res9: String =
```

```
(8) MapPartitionsRDD[622] at reduceByKey at <console>:49 []
```

```
| ShuffledRDD[615] at partitionBy at <console>:48 []
```

```
|     CachedPartitions: 8; MemorySize: 1754.8 MB; DiskSize: 0.0 B
```

## Operations that *might* cause a shuffle

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

## Avoiding a Network Shuffle By Partitioning

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

*Can you think of an example?*

## Avoiding a Network Shuffle By Partitioning

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

*Can you think of an example?*

### 2 Examples:

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

## Shuffles Happen: Key Takeaways

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

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

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

# Closures

## Closures are central to RDDs.

- ▶ Passed to most transformations.
- ▶ Passed to *some* actions (like reduce and foreach).

# Closures

## Closures are central to RDDs.

- ▶ Passed to most transformations.
- ▶ Passed to *some* actions (like reduce and foreach).

However, they can also cause issues that are *specific to distribution* (but would not be problematic with parallel collections, say)

# Closures

## Closures are central to RDDs.

- ▶ Passed to most transformations.
- ▶ Passed to *some* actions (like reduce and foreach).

However, they can also cause issues that are *specific to distribution* (but would not be problematic with parallel collections, say)

Two main issues:

1. Serialization exceptions at run time when closures are not serializable.
2. Closures that are “too large.”

# Closures

## Closures are central to RDDs.

- ▶ Passed to most transformations.
- ▶ Passed to *some* actions (like reduce and foreach).

However, they can also cause issues that are *specific to distribution* (but would not be problematic with parallel collections, say)

Two main issues:

1. Serialization exceptions at run time when closures are not serializable.
2. **Closures that are “too large.”**

## Closure Troubles: Example

```
class MyCoolApp {  
  val repos: RDD[Repository] = ... // repositories on GitHub (many!)  
  val team: Map[String, List[String]] = ... // maps username to skills  
  // GitHub repos that users in "team" map contribute to  
  def projects(): Array[Repository] = {  
    val filtered = repos.filter { repo =>  
      team.exists(user => repo.contributors.contains(user))  
    }  
    filtered.collect()  
  }  
}
```

## Closure Troubles: Example

```
class MyCoolApp {  
  val repos: RDD[Repository] = ... // repositories on GitHub (many!)  
  val team: Map[String, List[String]] = ... // maps username to skills  
  // GitHub repos that users in "team" map contribute to  
  def projects(): Array[Repository] = {  
    val filtered = repos.filter { repo =>  
      team.exists(user => repo.contributors.contains(user))  
    }  
    filtered.collect()  
  }  
}
```

*What happens when you run this?*

## Closure Troubles: Example

*What happens when you run this?*

## Closure Troubles: Example

*What happens when you run this?*

**java.io.NotSerializableException**

## Closure Troubles: Example

*What happens when you run this?*

**java.io.NotSerializableException**

**Why?**

## Closure Troubles: Example

*What happens when you run this?*

**java.io.NotSerializableException**

**Why?**

Let's have a look at the closure passed to the RDD:

```
val filtered = repos.filter { repo =>
  team.exists(user => repo.contributors.contains(user))
}
```

## Closure Troubles: Example

*What happens when you run this?*

**java.io.NotSerializableException**

**Why?**

Let's have a look at the closure passed to the RDD:

```
val filtered = repos.filter { repo =>
  team.exists(user => repo.contributors.contains(user))
}
```

Is this closure serializable?

## Closure Troubles: Example

*What happens when you run this?*

**java.io.NotSerializableException**

**Why?**

Let's have a look at the closure passed to the RDD:

```
val filtered = repos.filter { repo =>
  team.exists(user => repo.contributors.contains(user))
}
```

Is this closure serializable?

*It should be:* it only captures the “team” map. Map[String, List[String]] is serializable in Scala.

## Closure Troubles: Example

*What happens when you run this?*

**java.io.NotSerializableException**

**Why?**

Let's have a look at the closure passed to the RDD:

```
val filtered = repos.filter { repo =>
  team.exists(user => repo.contributors.contains(user))
}
```

Is this closure serializable?

*It should be:* it only captures the “team” map. `Map[String, List[String]]` is serializable in Scala.

In reality: closure is **not serializable!**

## Closures: Variable Capture

*A closure is serializable if...*

## Closures: Variable Capture

*A closure is serializable if...*

*...all captured variables are serializable.*

## Closures: Variable Capture

*A closure is serializable if...*

*...all captured variables are serializable.*

```
val filtered = repos.filter { repo =>
  team.exists(user => repo.contributors.contains(user))
}
```

*What are the captured variables?*

## Closures: Variable Capture

*A closure is serializable if...*

*...all captured variables are serializable.*

```
val filtered = repos.filter { repo =>  
    team.exists(user => repo.contributors.contains(user))  
}
```

*What are the captured variables?*

Just team.

## Closures: Variable Capture

*A closure is serializable if...*

*...all captured variables are serializable.*

```
val filtered = repos.filter { repo =>
  team.exists(user => repo.contributors.contains(user))
}
```

*What are the captured variables?*

Just team.

**Wrong!**

## Closures: Variable Capture

*A closure is serializable if...*

*...all captured variables are serializable.*

**...isn't true. Why?**

## Closures: Variable Capture

*A closure is serializable if...*

*...all captured variables are serializable.*

**...isn't true. Why?**

Instead of `team`, it is `this` (of type `MyCoolApp`) which is captured:

```
val filtered = repos.filter { repo =>
  this.team.exists(user => repo.contributors.contains(user))
}
```

## Closures: Variable Capture

*A closure is serializable if...*

*...all captured variables are serializable.*

**...isn't true. Why?**

Instead of `team`, it is `this` (of type `MyCoolApp`) which is captured:

```
val filtered = repos.filter { repo =>
  this.team.exists(user => repo.contributors.contains(user))
}
```

However, this is not serializable. `MyCoolApp` does not extend the marker interface `Serializable`.

## Closure Trouble: Solution 1

Make a local copy of team. No more accidental capturing of MyCoolApp.

It should be written like this:

```
val localTeam = team
val filtered = repos.filter { repo =>
  localTeam.keys.exists(user => repo.contributors.contains(user))
}
```

With localTeam, this is no longer captured. **Now it's serializable.**

## Closure Trouble: Big Closures

Let's assume that this and everything within it (MyCoolApp) is serializable.

### **Problem:**

It could be silently capturing, serializing, and sending over the network, some huge pieces of captured data. Typically the only hint of this occurring is high memory usage and long run times.

**Note: this is a real problem which could appear in your programming assignments! If you're using too much memory, and if performance is slow, make sure you're not accidentally capturing large enclosing objects!**

## Shared Variables

Normally, when a function passed to a Spark operation (such as map or reduce) is executed on a remote cluster node, it works on separate copies of all the variables used in the function.

These variables are copied to each machine, and no updates to the variables on the remote machine are propagated back to the driver program.

However, Spark does provide two limited types of shared variables for two common usage patterns:

1. **Broadcast variables**
2. **Accumulators**

## Broadcast Variables

Let's revisit the closure from a few slides ago:

```
val localTeam = team
val filtered = repos.filter { repo =>
  localTeam.keys.exists(user => repo.contributors.contains(user))
}
```

1. What if `localTeam/team` is a Map of thousands of elements?
2. What if several operations require it?

## Broadcast Variables

Let's revisit the closure from a few slides ago:

```
val localTeam = team
val filtered = repos.filter { repo =>
  localTeam.keys.exists(user => repo.contributors.contains(user))
}
```

1. What if `localTeam/team` is a Map of thousands of elements?
2. What if several operations require it?

**This is the ideal use-case for *broadcast variables*.**

## Broadcast Variables

Broadcast variables:

- ▶ allow the programmer to keep a read-only variable cached on each machine rather than shipping a copy of it with tasks.

They can be used, for example, to give every node a copy of a large input dataset in an efficient manner.

Spark also distributes broadcast variables using efficient broadcast algorithms to reduce communication cost.

## Broadcast Variables

To make `localTeam/team` a broadcast variable, all we have to do is:

```
val broadcastTeam = sc.broadcast(team)
```

We can then use it in our closures without having to ship it over the network multiple times!

*(Its value can be accessed by calling the value method)*

```
val filtered = repos.filter { repo =>
  broadcastTeam.value.keys.exists(user => repo.contributors.contains(user))
}
```

# Accumulators

Accumulators:

- ▶ are variables that are only “added” to through an associative operation and can therefore be efficiently supported across nodes in parallel.
- ▶ provide a simple syntax for aggregating values from worker nodes back to the driver program.

They can be used to implement counters (as in MapReduce) or sums.

**Out of the box, only numeric accumulators are supported in Spark.**

But it's possible to add support for your own types with a bit of effort.

## Accumulators: Example

```
val badRecords = sc.accumulator(0)
```

```
val badBytes = sc.accumulator(0.0)
```

```
records.filter(r => {
```

```
  if (isBad(r)) {
```

```
    badRecords += 1
```

```
    badBytes += r.size
```

```
    false
```

```
  } else {
```

```
    true
```

```
  }
```

```
}).save(...)
```

```
printf("Total bad records: %d, avg size: %f\n",
```

```
  badRecords.value, badBytes.value / badRecords.value)
```

## Accumulators

Accumulators can appear both in transformations and actions.

*What about fault tolerance? What happens to an accumulator when a node dies and must be restarted?*

# Accumulators

Accumulators can appear both in transformations and actions.

*What about fault tolerance? What happens to an accumulator when a node dies and must be restarted?*

## **Accumulators and fault tolerance:**

- ▶ **Actions:** Each task's update is applied to each accumulator only once.
- ▶ **Transformations:** An accumulator update within a transformation can occur more than once. E.g., when an RDD is recomputed from its lineage, it can update the accumulator. *Should only be used for debugging in transformations.*