ÉCOLE POLYTECHNIQUE
FÉDÉRALE DE LAUSANNE

Optimizing with Partitioners

Big Data Analysis with Scala and Spark

Heather Miller

Optimizing with Partitioners

We saw in the last session that Spark makes a few kinds of partitioners available out-of-the-box to users:

- **hash partitioners** and
- **range partitioners**.

We also learned what kinds of operations may introduce new partitioners, or which may discard custom partitioners.

However, we haven't covered *why* someone would want to repartition their data.

Optimizing with Partitioners

We saw in the last session that Spark makes a few kinds of partitioners available out-of-the-box to users:

- **hash partitioners** and
- **range partitioners**.

We also learned what kinds of operations may introduce new partitioners, or which may discard custom partitioners.

However, we haven't covered *why* someone would want to repartition their data.

Partitioning can bring substantial performance gains, especially in the face of shuffles.

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?

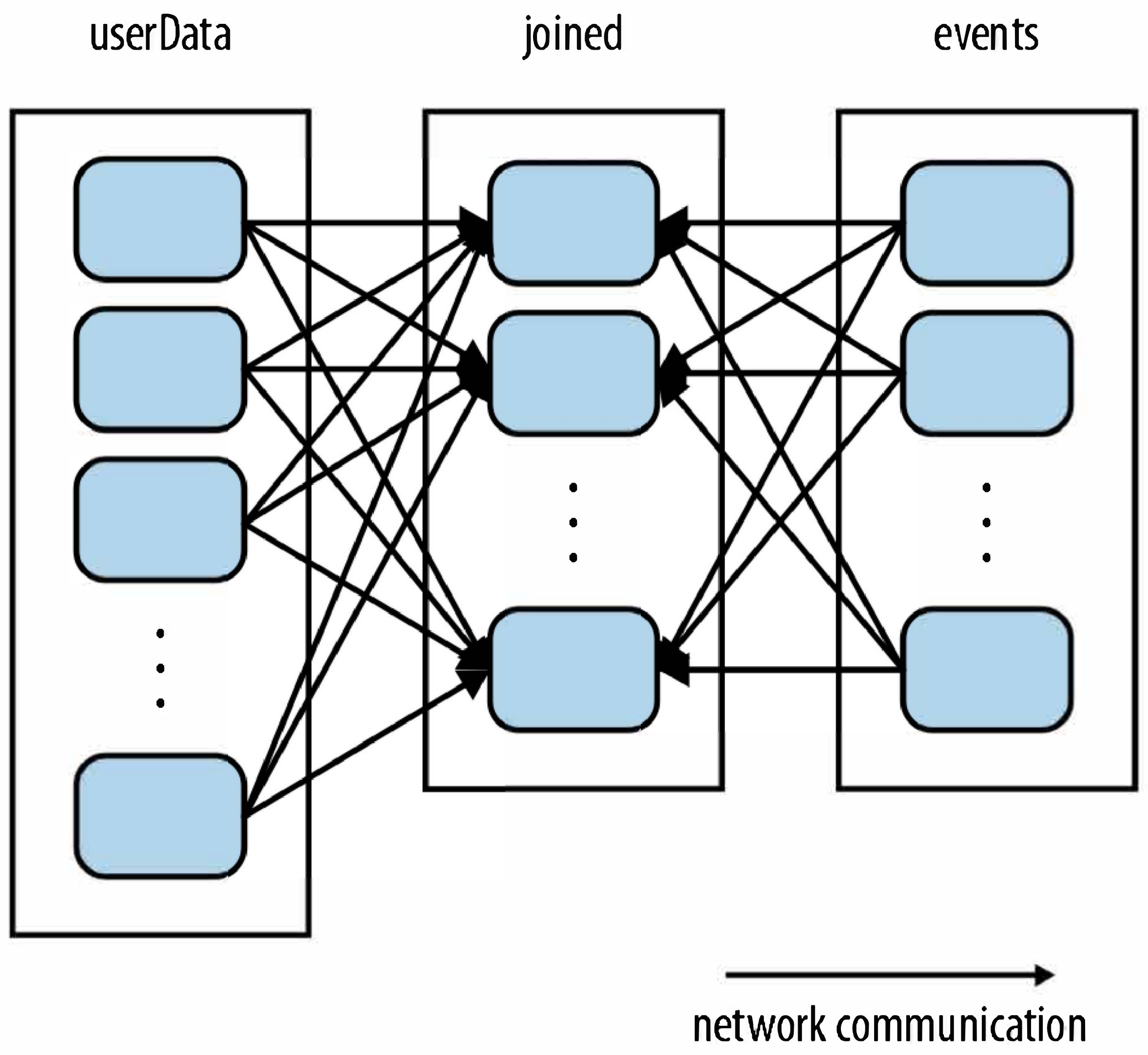
Partitioning Data: `partitionBy`, Another Example

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:

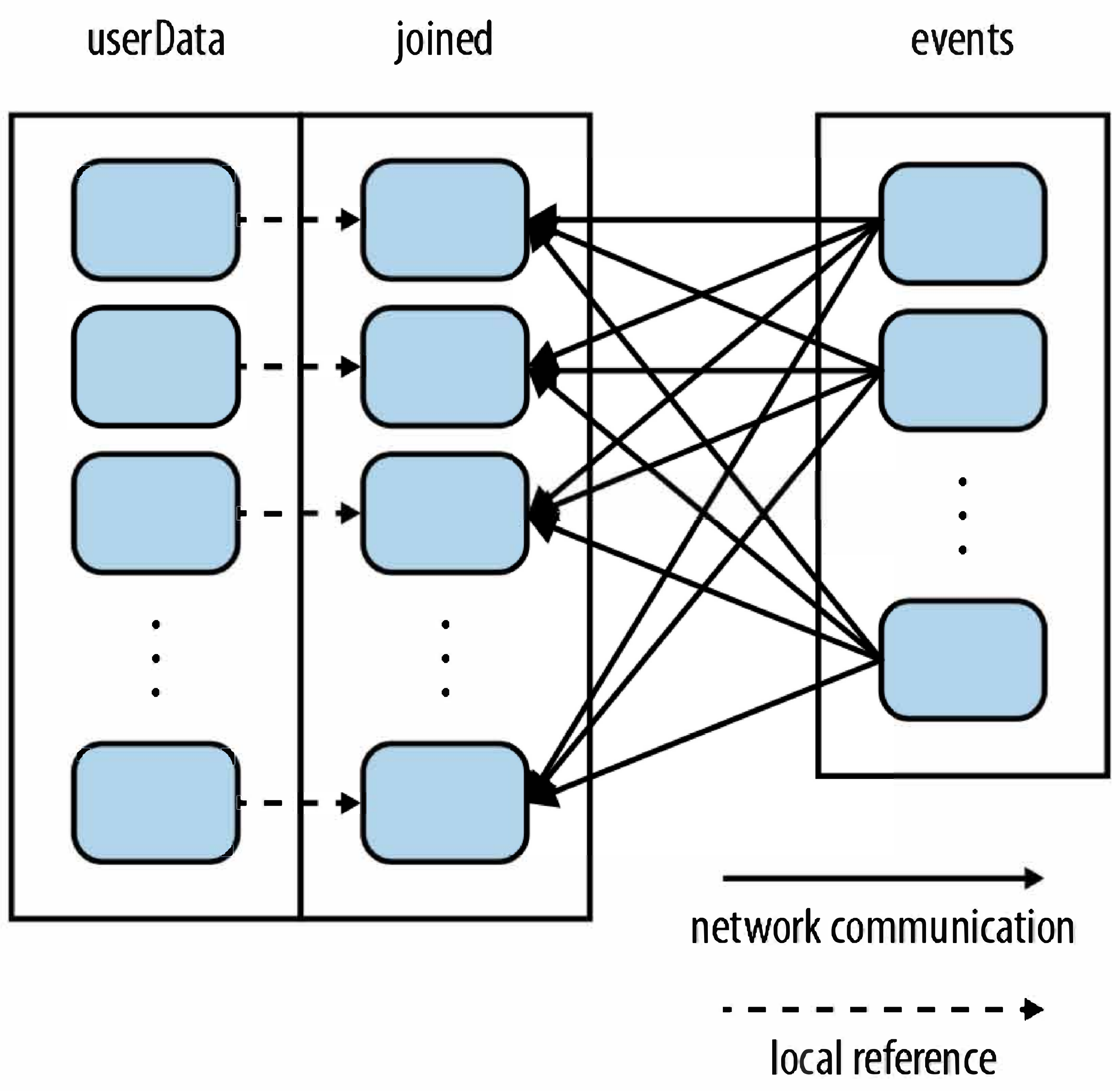
```
val userData = sc.sequenceFile[UserID, UserInfo]("hdfs://...")
                 .partitionBy(new HashPartitioner(100)) // Create 100 partitions
                 .persist()
```

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.