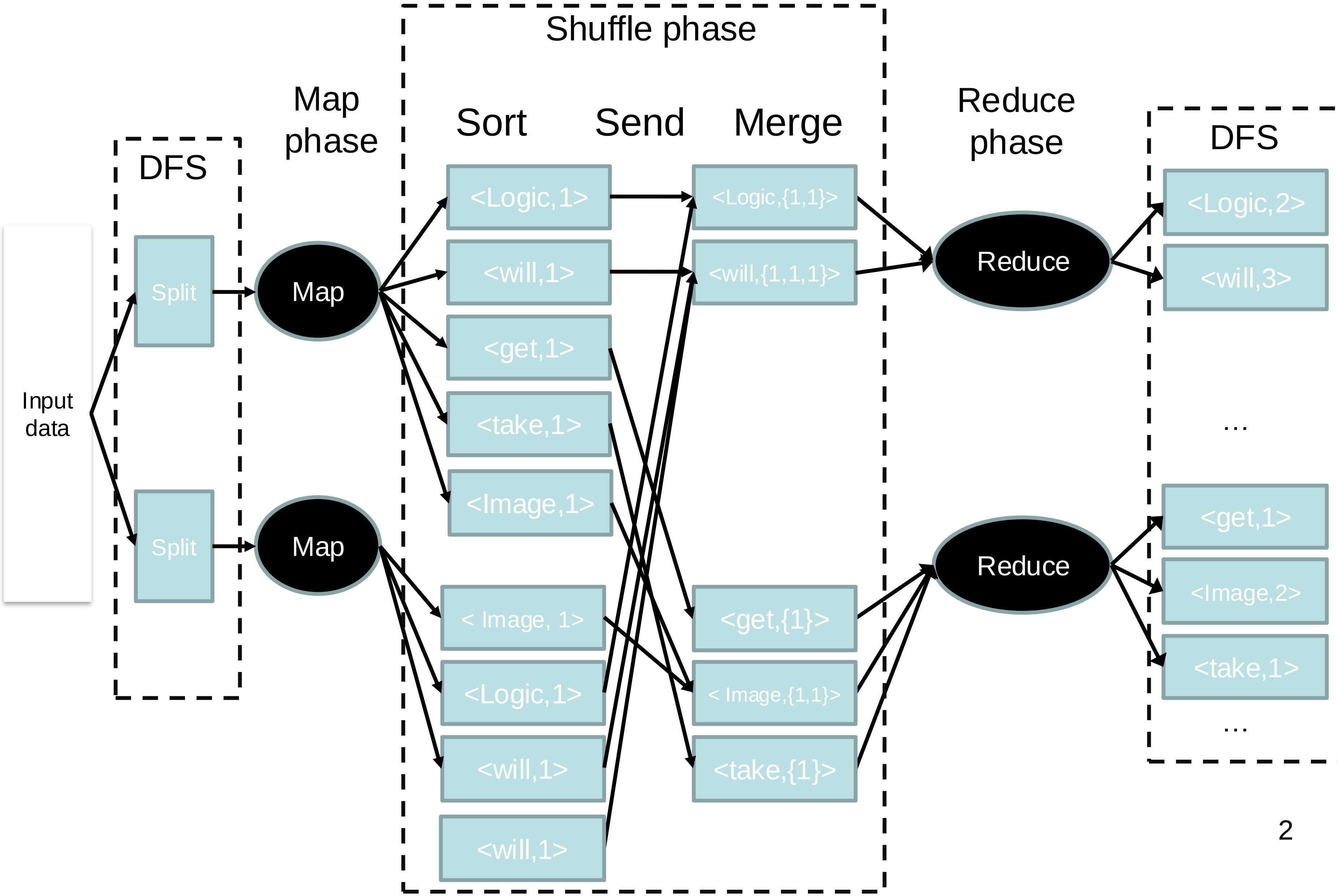
Programming for Data Science at Scale

Optimising Distributed Data Processing

THE UNIVERSITY of EDINBURGH

Amir Noohi, Fall 2024

MapReduce – under the hood



What is shuffling?

What happens when you do a `groupBy` or a `groupByKey`?

```
// Create an RDD of (Grade, Student) pairs
val students = sc.parallelize(List(
  ("A", "Alice"),
  ("B", "Bob"),
  ("A", "Adam"),
  ("C", "Charlie"),
  ("B", "Ben")
))

// Group students by their grade
val groupedStudents = students.groupByKey()

// The output of groupByKey is a ShuffledRDD
// Type of groupedStudents: RDD[(String, Iterable[String])] =
ShuffledRDD[4] at groupByKey at <console>:24
```

move data from one node to another to be "grouped with" its key.

Shuffling is **expensive** because:

- Network I/O (moving data between nodes).
- Disk I/O when data is too large to fit in memory.
- Serialisation and deserialisation of data.

Example of Shuffling

We have a list of three ATMs from which customers withdraw money. Now, we want to calculate how much each customer has withdrawn in total.

```scala
// Define a case class for ATM withdrawals
case class ATMWithdrawal(customerId: Int, atmId: Int, amount: Double)

// Create an RDD of customer withdrawals from different ATMs, distributed across 3
partitions
val withdrawals = sc.parallelize(List(
  ATMWithdrawal(1, 101, 200.0),  // Partition 1
  ATMWithdrawal(2, 102, 150.0),  // Partition 1
  ATMWithdrawal(3, 103, 300.0),  // Partition 2
  ATMWithdrawal(1, 101, 100.0),  // Partition 2
  ATMWithdrawal(2, 102, 50.0),   // Partition 3
  ATMWithdrawal(3, 103, 400.0)   // Partition 3
), 3)  // The data is split across 3 partitions
```

Example of Shuffling

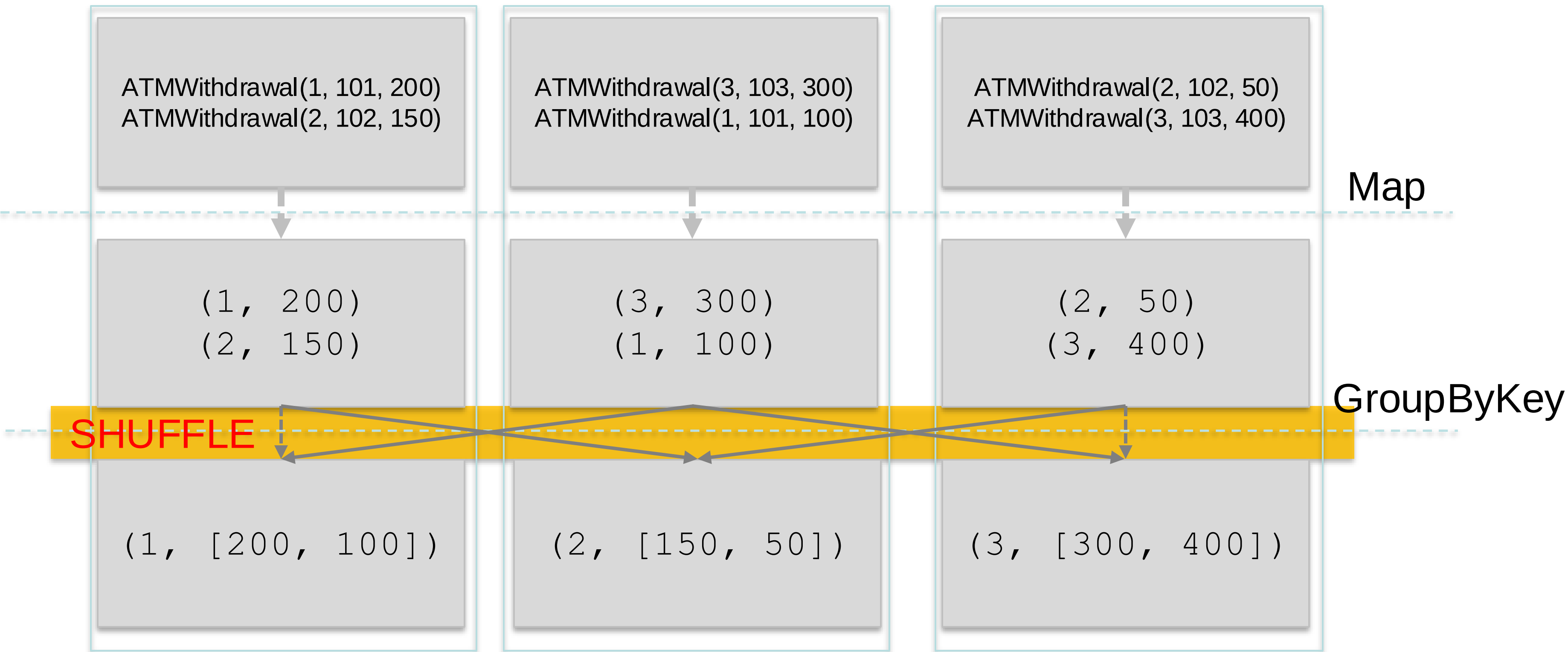
What is the solution?

```
// Group withdrawals by customerId and calculate the total amount per customer
val totalWithdrawalsPerCustomer = withdrawals
  .map(w ⇒ (w.customerId, w.amount))    // Convert to (customerId, amount) tuples
  .groupByKey()                         // Group by customerId (Shuffle occurs here)
  .mapValues(amounts ⇒ amounts.sum)      // Sum all amounts for each customer
```

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

Which data needs to be moved between nodes?

Example of Shuffling



Can we make it better?

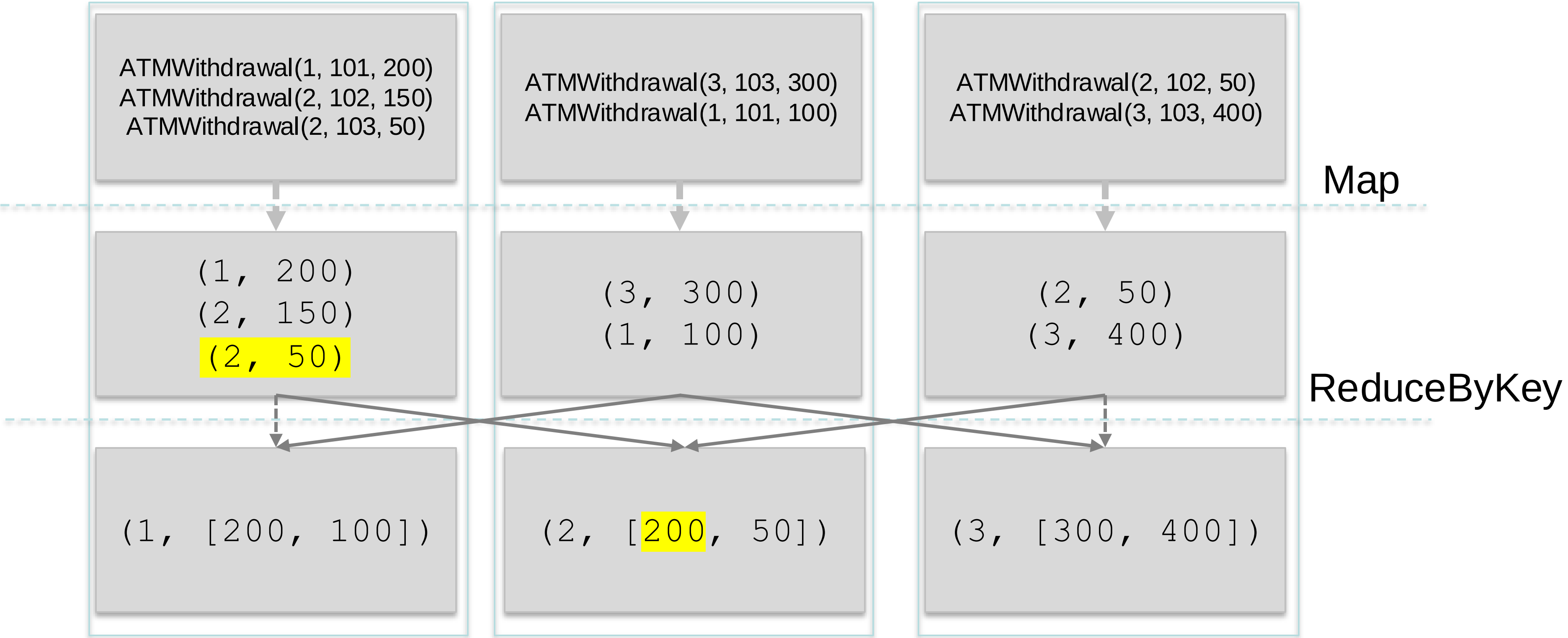
Reduce Instead of Grouping

`reduceByKey` combines `groupByKey` and reduction into one operation

Key Advantage: **local aggregation**

```
val totalWithdrawalsPerCustomer = withdrawals
  .map(w ⇒ (w.customerId, w.amount))  // (customerId, amount)
  .reduceByKey(_ + _)                   // Sum amounts per customer
```

Example of Shuffling



When will shuffle occur?

1. The return type of certain transformations:

```
org.apache.spark.rdd.RDD[(String, Int)]= ShuffledRDD[1104]
```

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

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

res9: String=
(8) MapPartitionsRDD[1104] at reduceByKey at <console>:49 []
                     | ShuffledRDD[l6151J at partitionBy at <console>:48 []
                     | CachedPartitions: 8; MemorySize: 1754.8 MB; DiskSize: 0.0 B
```

Where else shuffling occurs?

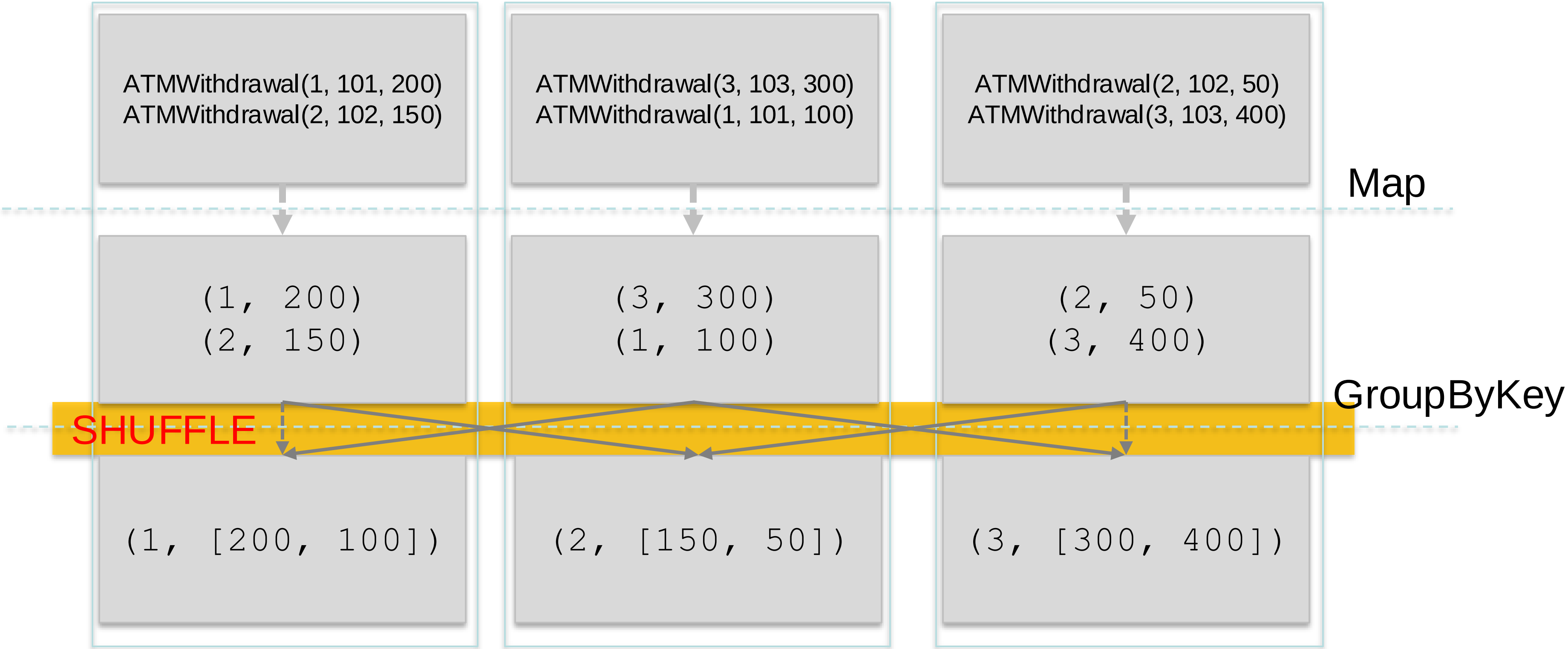
1. `groupByKey()`:
   - Spark needs to move all records with the same key to the same partition.
2. `reduceByKey()`:
   - Shuffling occurs **after local aggregation** when Spark needs to move partial sums between partitions to calculate the final result.
3. `join()`:
   - Spark must align keys from two RDDs.
4. `distinct()`:
   - ensure that duplicate records across partitions are compared and removed.
5. `sortByKey()`:
   - Spark needs to globally sort data across all partitions.
6. `repartition()`:
   - redistributing data into a different number of partitions.

Runtime of Shuffling

```
> 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
```

Example of Shuffling



What is Partition?

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.

Key Properties:

- Partitions **never span multiple machines**; all data in a partition stays on one machine.
- Each machine in the cluster contains **one or more partitions**.
- The **number of partitions** is configurable (default = total number of cores across executor nodes).

But how does Spark know which key to put on which machine?

Types of Partitioning:

1. Hash Partitioning
2. Range Partitioning

Hash Partitioning

- **Hashing**: Spark applies a hash function to `customerId` (e.g., 1, 2, 3) to determine the partition.

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

- **Partitioning**: Data with the same hash value goes to the same partition.
  - Different `customerId`s go to different partitions based on their hash.
- **Result**: All records for a specific key are grouped into a single partition based on the hash function, ensuring **efficient distribution**

Range Partitioning

- Spark **sorts the keys** and divides them into ranges.
- Each partition holds a **specific range of keys** (e.g., 1-100 in one partition, 101-200 in another).
- **Efficient for ordered data** or when you need to process data within specific key ranges.

Example:
Withdrawals Data: If customerIds range from 1-1000, Spark splits this into partitions like:

- **Partition 1**: customerId 1-100
- **Partition 2**: customerId 101-200
- **Partition 3**: customerId 201-300

Partitioning Data

There are two ways to create RDDs with specific partitioning:

1. Call `partitionBy` on an RDD, providing an explicit Partitioner.
   - Apply `partitionBy()` and provide an explicit **Partitioner** (e.g., Hash or Range).

```
// Example with HashPartitioner
val partitionedRDD = rdd.partitionBy(new HashPartitioner(numPartitions))

// Example with RangePartitioner
val rangePartitionedRDD = rdd.partitionBy(new RangePartitioner(numPartitions, rdd))
```

2. Using transformations that return RDDs with specific partitioners

```
val reducedRDD = rdd.reduceByKey(_ + _) // Automatically hash partitioned
```

Persisting Partitioned Data

Problem:

- After `partitionBy()`, Spark **re-shuffles and recomputes** the entire RDD every time you perform an action (e.g., `count()`, `collect()`).

Solution: Persist!

- `persist()` stores the RDD in memory (or disk) after the first computation.

```
// Without persist - recomputes partitioning every time
val partitionedRdd = rdd.partitionBy(new HashPartitioner(100))
partitionedRdd.count()   // Recomputes partitionBy and counts
partitionedRdd.collect() // Recomputes partitionBy again

// With persist - avoids recomputation
partitionedRdd.persist()
partitionedRdd.count()   // Computes partitionBy and persists
partitionedRdd.collect() // Reuses the persisted data, no recomputation
```

Partitioner Inheritance

1. **Partitioner from Parent RDD**
   - Pair RDDs resulting from transformations on a partitioned RDD **inherit the partitioner** (usually Hash) from the parent RDD.

```
val transformedRDD = parentRDD.mapValues( ... ) // Uses the same partitioner as parentRDD
```

2. **Automatically-set Partitioners**

   Some operations **automatically apply partitioners** when it makes sense:
   - `sortByKey`: Uses a **RangePartitioner** by default.
   - `groupByKey`: Uses a **HashPartitioner** by default.

```
val sortedRDD = rdd.sortByKey()   // RangePartitioner is used
val groupedRDD = rdd.groupByKey() // HashPartitioner is used
```

Automatic Partitioners

Certain operations on Pair RDDs **retain and propagate** the partitioner from the parent RDD:

1. **Cogroup**
2. **groupWith**
3. **Join, leftOuterJoin,** r**ightOuterJoin**
4. **groupByKey**
5. **reduceByKey**, **foldByKey**,**combineByKey**
6. **partitionBy**
7. **sortmapValues** (if parent has a partitioner)
8. **flatMapValues** (if parent has a partitioner)
9. **filter** (if parent has a partitioner)

All other operations will produce a result without a partitioner.

Partitioning Example

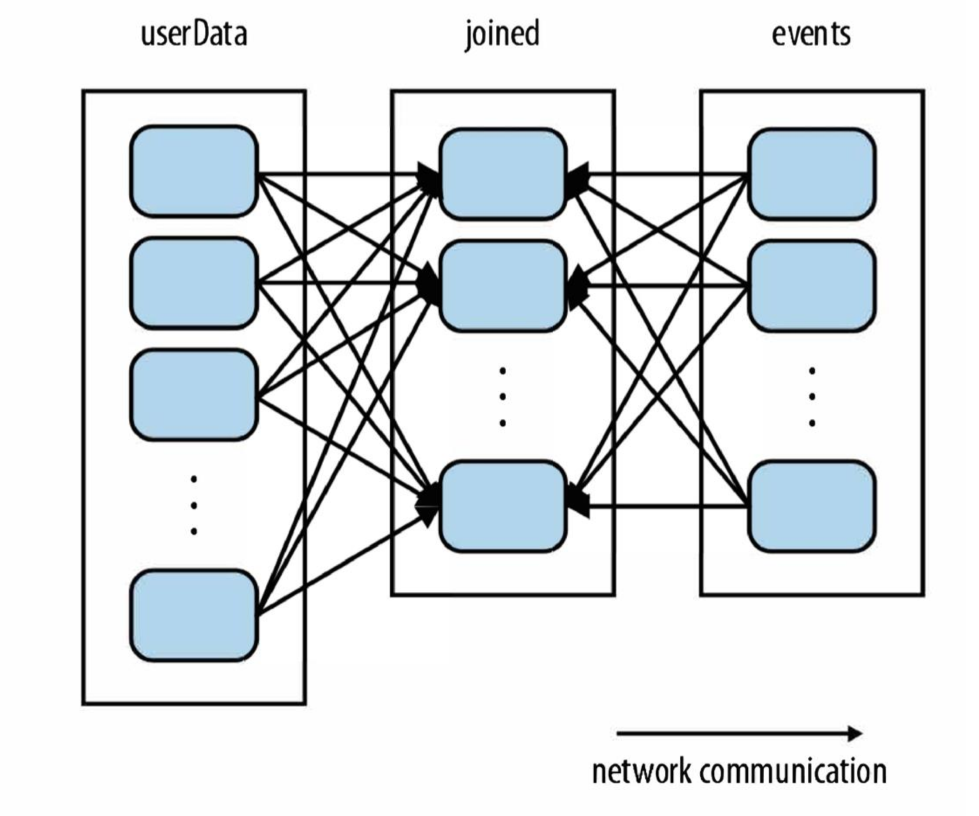
```
val sc = new SparkContext( ... )
val userData = sc.sequenceFile[UserID, Userlnfo]("hdfs:// ... ").persist()


def processNewlogs(logFileName: String) {
  val events = sc.sequenceFile[UserID, Linklnfo](logFileName)
  val joined = userData.join(events) //ROD of (UserID, (Userlnfo, Linklnfo))
  val offTopicVisits = joined.filter {
    case (userld, (userlnfo, linklnfo)) ⇒ //Expand the tuple
        !userlnfo.topics.contains(linklnfo.topic)
  }.count()
  println(''Number of visits to non-subscribed topics: '' + offTopicVisi ts)
}
```

Is this OK?

Partitioning Example

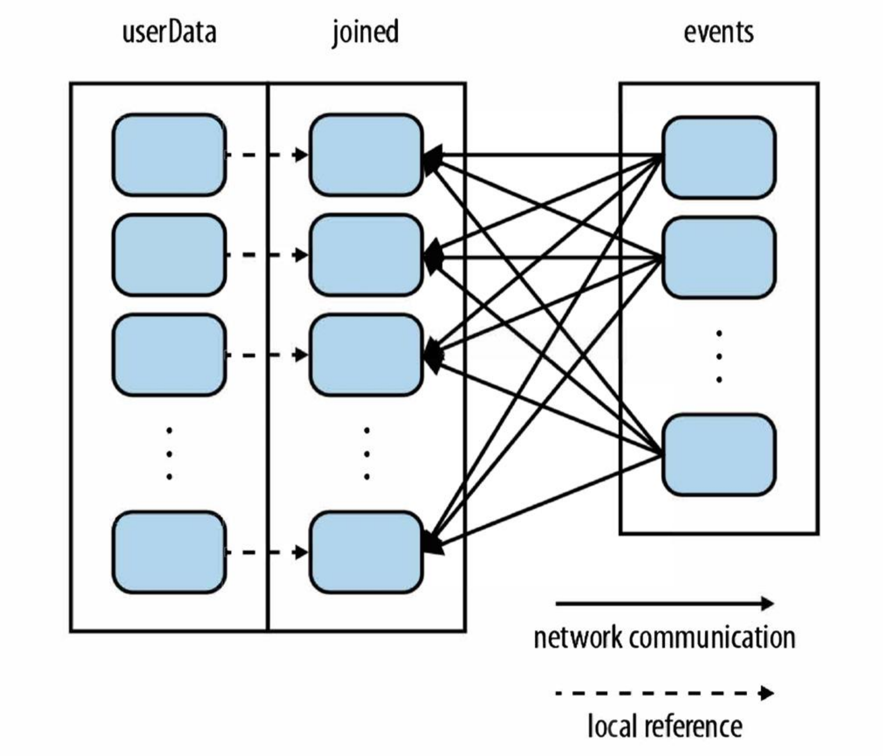
It will be very inefficient!

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



Explicit Partitioning Example

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



QUESTIONS?