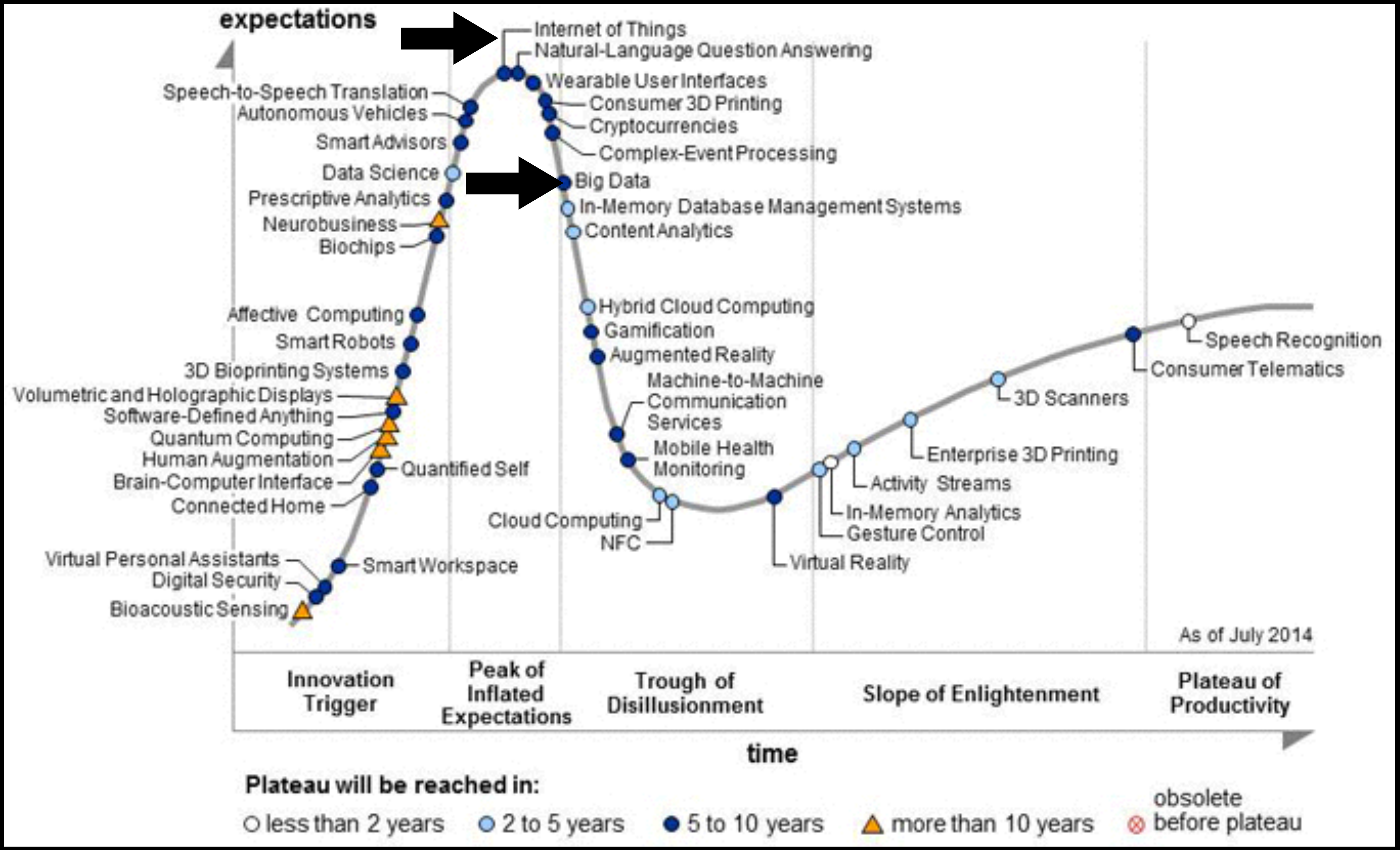
Distributed Computing

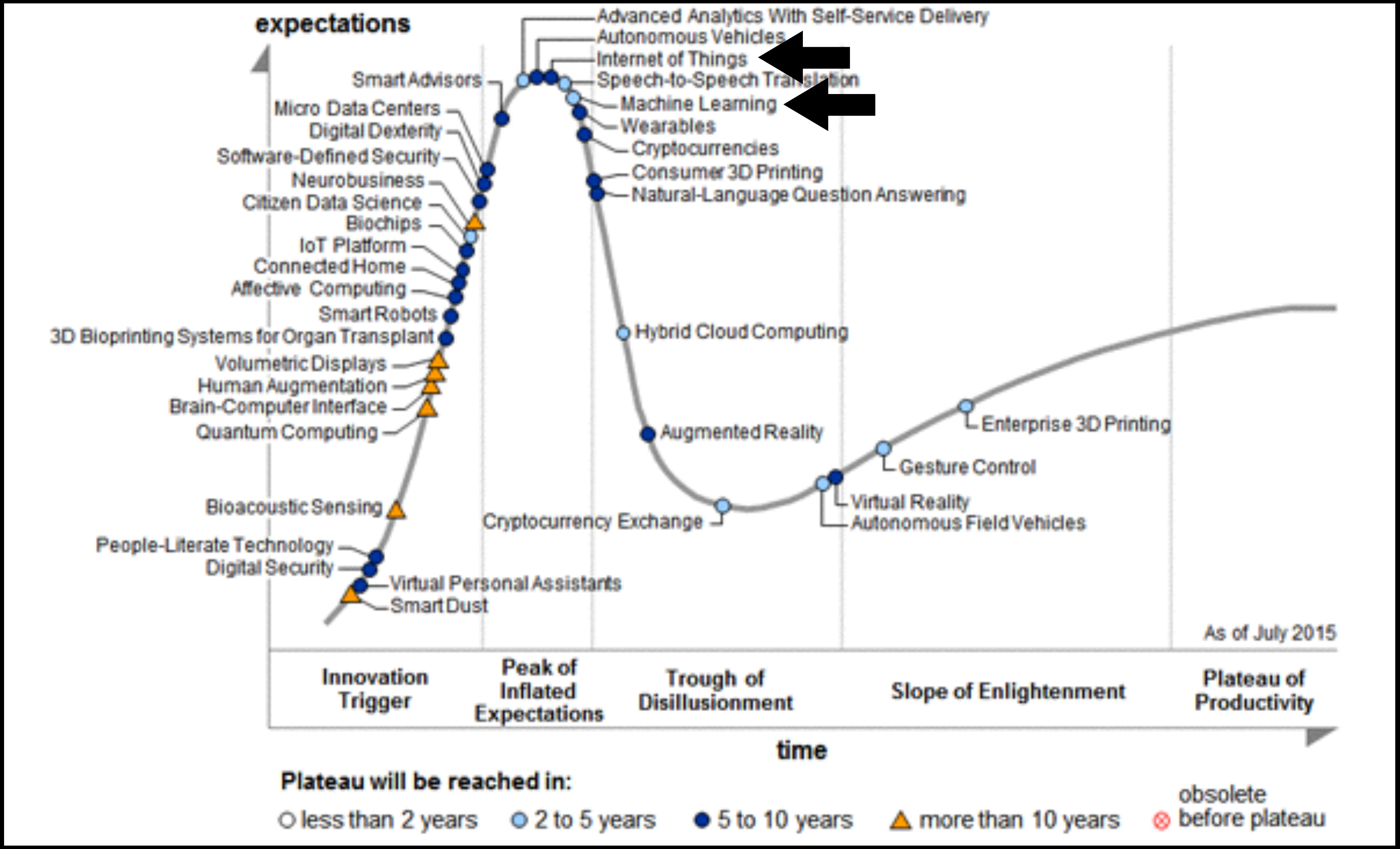
using Apache Spark

Technology Trends



Gartner’s hype cycle

2014



Gartner's hype cycle

2015

Big Data

- Walmart collects 2.5 petabytes every hour
- They found that Strawberry pop-tarts sales increased by 7 times before a Hurricane.
- Now they place all the Strawberry pop-tarts at the checkouts before hurricanes.
- They own 250 node cluster to do this analysis!

http://www.dezyre.com/article/how-big-data-analysis-helped-increase-walmart-s-sales-turnover/109

Challenges

- Challenges of Big data is massive size of data
- Data may not fit in the memory of computer
- Splitting data across machines introduces more complications (imagine calculating average of list of numbers distributed on 10 machines)

Machine Learning

- Facebook recommends items
  - Which posts appear in your timeline
  - suggesting friends
- 100B rating, 1B users, millions of items

https://code.facebook.com/posts/861999383875667/recommending-items-to-more-than-a-billion-people/

Challenges

- Same challenges as big data

Internet of Things

- More ‘things’ are connected to Internet
- Tesla’s over the air software update
- Smart TVs (Android TV, Apple TV, etc)
- Smart Watches
- Health devices (Fitbit)

Challenges

- Traffic can’t be handled by one network card
- Processing must be as fast as possible, we need to take actions immediately

Conclusion

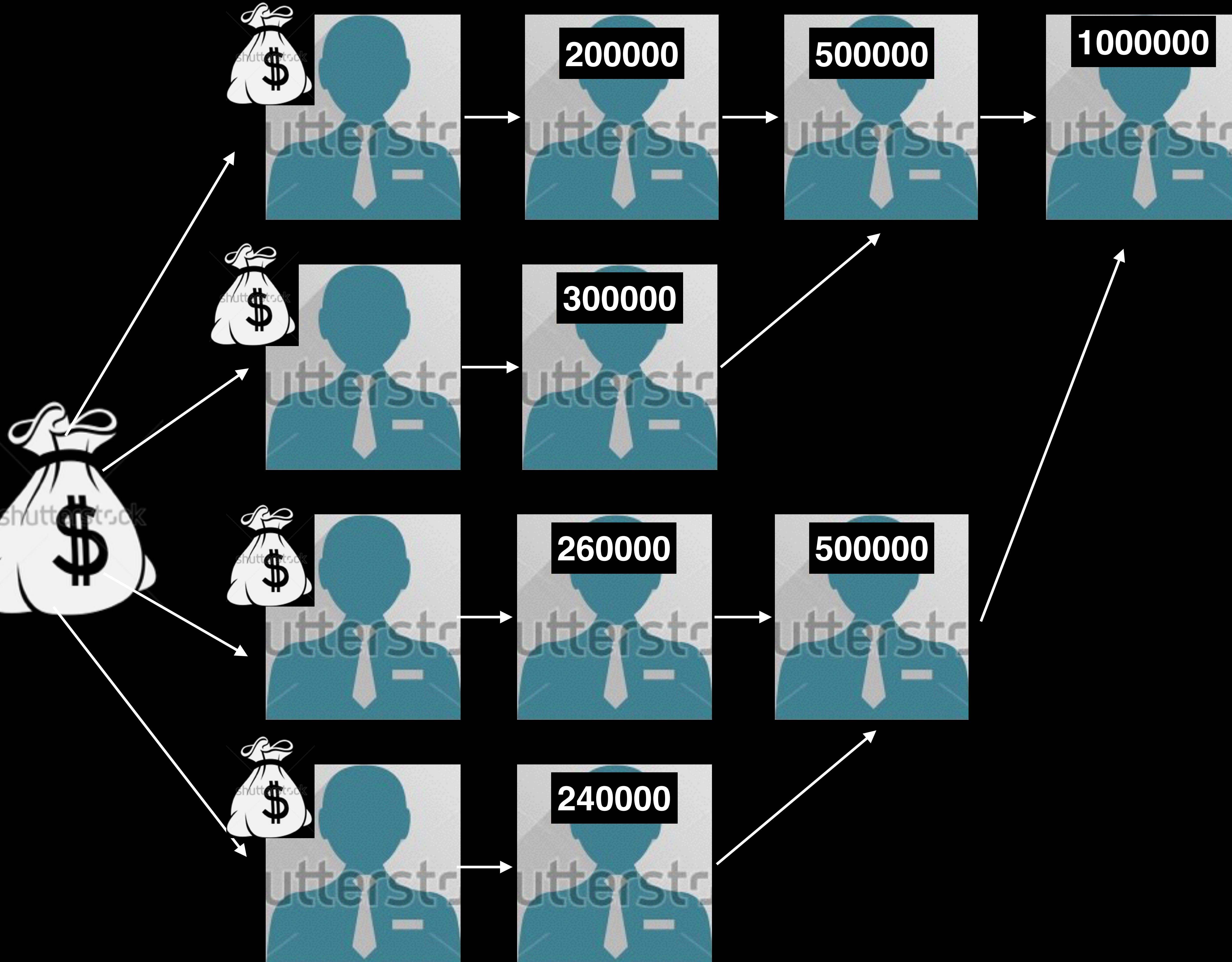
- We must scale to more than one machine
- Distributed programming has costs that we must pay to cope with requirements

Distributed Programming

- Counting a million dollars would take you about 3 hours.
- Luckily you can easily divide the work among your friends!

Counting Example

- Divide money among friends
- Count in parallel
- Add the counts to get the total count


200000
500000
1000000
300000
260000
500000
240000

Coding Time

- Download Spark

  http://www.eu.apache.org/dist/spark/spark-1.5.1/spark-1.5.1-bin-hadoop2.6.tgz

- Open Spark shell
  - bin/spark-shell ◀ Scala
  - bin/pyspark ◀ Python
- Notice “Started SparkUI at http://localhost:4040”
- Open the url in browser

```scala
val notes = List.fill(10000)((100, "USD"))

val distributedNotes = sc.parallelize(notes)

val values =
  distributedNotes.map(note => note._1)

values.reduce((v1, v2) => v1 + v2)
```

Create list of 10000 pairs

Transform the collection into new one

Executes and return the result

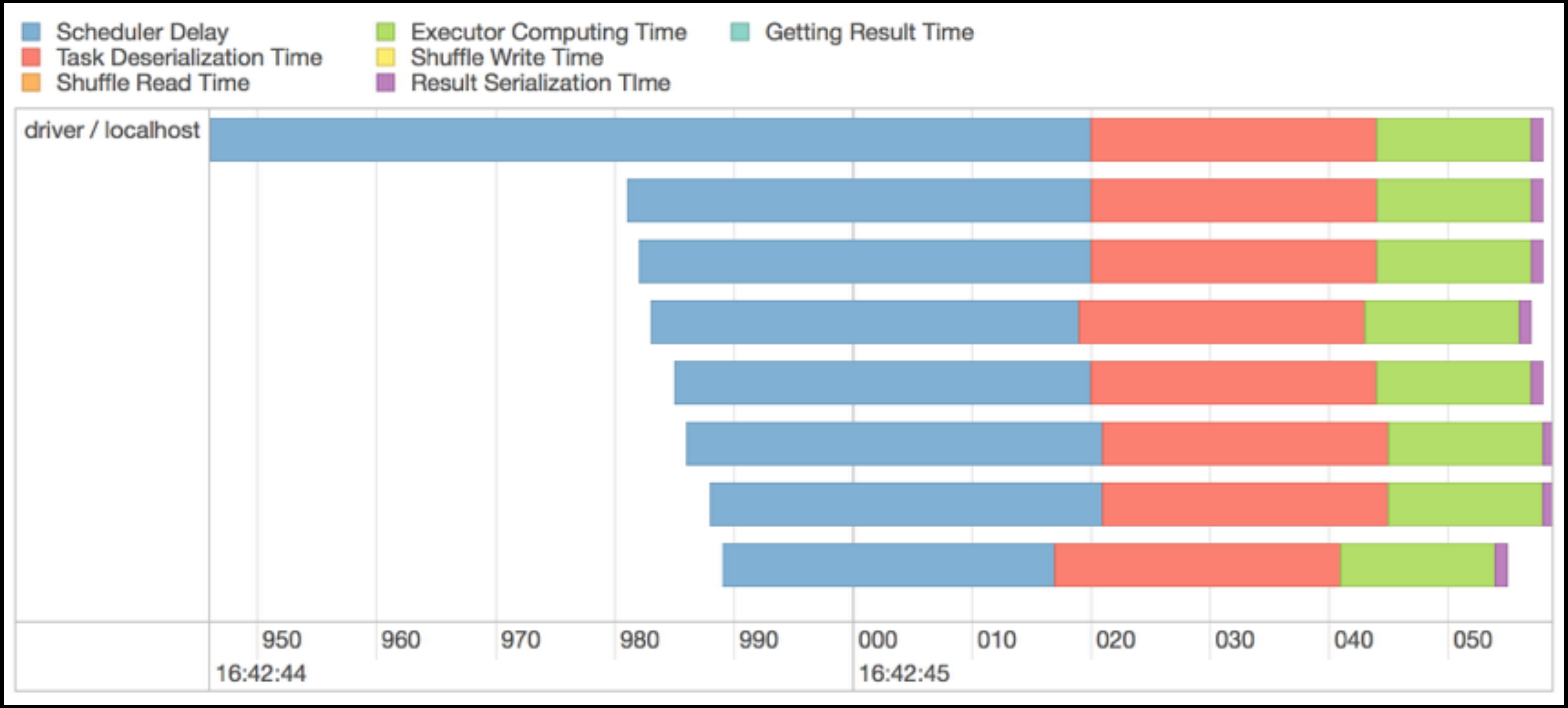
Create list of 10000 tuples

Transform collection into new one

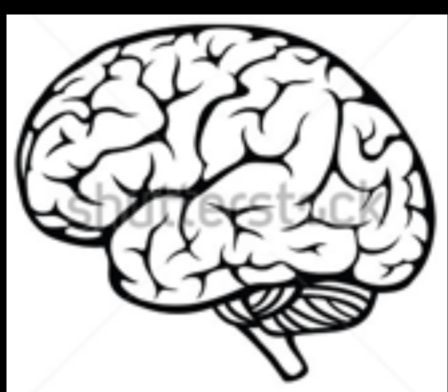
```python
notes = [(100, “USD”)] * 10000

distributedNotes = sc.parallelize(notes)

values = distributedNotes.map(lambda note: note[0])

values.reduce(lambda v1, v2: v1 + v2)
```

Executes and return the result

Spark UI


Scheduler Delay
Task Deserialization Time
Shuffle Read Time
Executor Computing Time
Shuffle Write Time
Result Serialization TIme
Getting Result Time
driver / localhost
950
960
970
980
990
000
010
020
030
040
050
16:42:44
16:42:45

How Spark Works



Driver

shell

Executor 1

Executor 2

...

Executor N

Executors can be running on same machine (local mode), or in different remote cluster without changing code

Driver

- Can be shell
- Or any program creates SparkContext instance
- Contains your logic
- Send ‘execution plan’ to executors and collects output

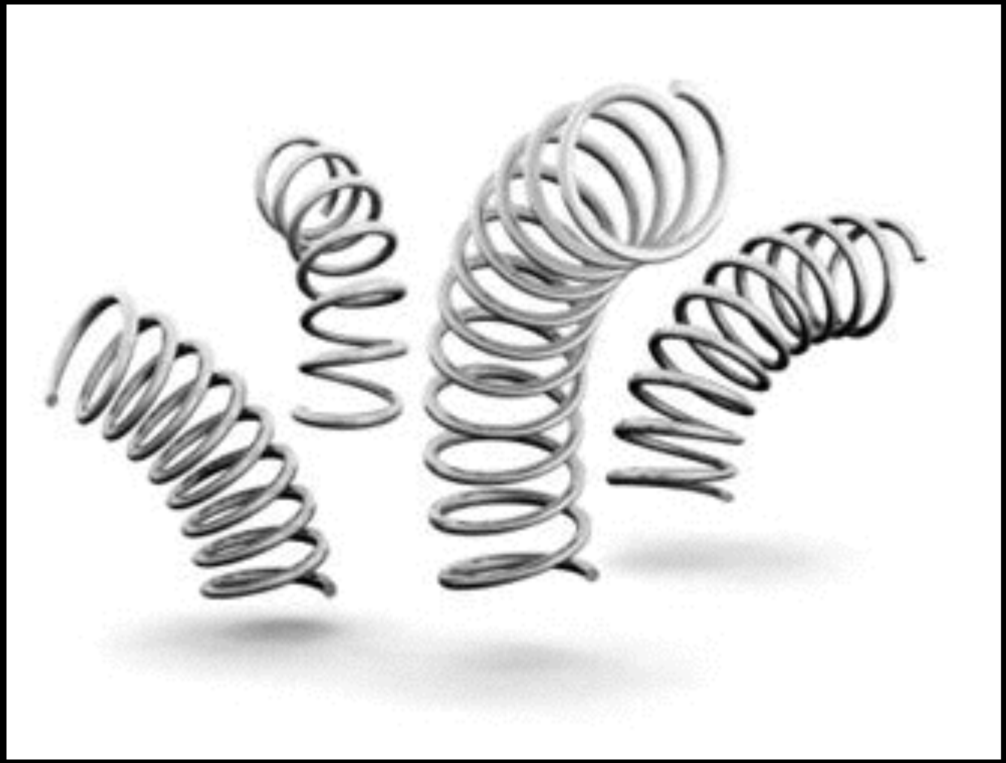
Java Driver Example

```
import org.apache.spark.SparkConf;
import org.apache.spark.SparkContext;

public class Test {

 public static void main(String[] args) {
   SparkContext sc =
      new SparkContext(new SparkConf().setAppName("Test"));


   …
 }

}
```

Executors

- They run the tasks (sent by driver) and return results back.
- They provide in-memory storage for RDDs
- Spark’s local mode runs executors on same machine as driver
- Usually executors run on dedicated machines

RDD



- Resilient Distributed Dataset
- Fault-tolerant collection of elements
- Can be operated in parallel
- Can be created programmatically
- Can be loaded from external storage (Database or text-file)
  sc.textFile(“data.txt”)

Creating RDDs

Programatically

```
sc.parallelize(data)
sc.range(1, 100)
```

Externally

```
sc.textFile("file.txt")
sc.textFile("data/*.txt")
sc.textFile("data")
```

Multiple files merged

Transformation

```
rdd1
  .map(….)
  .filter(….)
```

Traditional Algorithms

```
class Count {
 public static void main(String[] args) {
  int[] money = new int[10000] {100, …};
  int count = 0
  for (int i = 0; i <= money.length; i++) {
    count += money[i]
  }
  System.out.println(count)
 }
}
```

```
money = [100] * 10000
count = 0
for i in range(len(money)):
  count += money[i]

print(count)
```

- Developer is responsible for iterating on array

Traditional Algorithms

```
class Count {
 public static void main(String[] args) {
  int[] money = new int[10000] {100, …};
  int count = 0
  for (int note : money) {
    count += note
  }
  System.out.println(count)
 }
}
```

```
money = [100] * 10000
count = 0
for note in money:
  count += note

print(count)
```

- Developer is responsible for “collecting” result

Imperative Style

- Previous example is considered “imperative”

[1] **imperative**

SAVE POPULARITY

adjective | im·per·a·tive | \im-ˈper-ə-tiv, -ˈpe-rə-\

Cite! Share G+1 Tweet

: very important

grammar : having the form that expresses a command rather than a statement or a question

: expressing a command in a forceful and confident way

Imperative programming

- is a programming paradigm that uses statements that change program's state (variables)

- Developer responsible for specifying the steps needed to reach the answer

Declarative programming

- is a programming paradigm that expresses the logic without describing it’s control flow.
- **Functional programming** languages are declarative
- SQL is declarative language

Functional style

```
class Count {
 public static void main(String[] args) {
  int[] money = new int[10000] {100, …};
   int count = Arrays
    .stream(money)
    .reduce(0, (a, b) -> a + b))

  System.out.println(count)
 }
}
```

```
money = [100] * 10000
count = reduce(
  lambda a, b: a + b,
   money)

print(count)
```

- Logic and control flow are separated
- Developer is responsible specifying logic
- Language/Framework responsible for execution details

Scala in a syringe!

Values

```
scala> val x = 5
x: Int = 5

scala> x = 4
<console>:8: error: reassignment to val
```

- Similar to final values in Java and const in C#
- You can’t set reassign to something else

Values have types

```
scala> val x: Int = 5
x: Int = 5
OR
scala> val x = 5
x: Int = 5
```

- Scala can detect the type (Type inference)
- Both code snippets above has no difference

Functions

```
scala> def add(x: Int, y: Int): Int = x + y
add: (x: Int, y: Int)Int

scala> add(3, 4)
res17: Int = 7
```

- Scala can infer the return types of functions
- Parameters type can’t inferred (expect for specific cases)

Functions

```
def add(x: Int, y: Int) = x + y

def add(x: Int, y: Int) = {
  val sum = x + y
  sum
}
```

- Last expression in function’s body is the return

Functions

```
scala> def five() = 5
five: ()Int

scala> five()
res15: Int = 5

scala> five
res16: Int = 5
```

- Functions with no parameters can be called without parenthesis

Anonymous Functions

```
scala> (x: Int, y: Int) => x + y
res18: (Int, Int) => Int = <function2>
```

- Similar to lambda in python
- Short syntax to create function

Anonymous Functions

```
(1 to 100).map(n => n/2).reduce((n1, n2) => n1 + n2)
```

- Usually used to send functions as parameters
- Scala can infer arguments’ types here

Anonymous Functions

```
(1 to 100).map(_ / 2).reduce(_ + _)
```

- Shorter syntax to create function!

Pair

```
scala> (1, "one")
res22: (Int, String) = (1,one)

scala> 1 -> "one"
res23: (Int, String) = (1,one)

scala> Pair(1,"one")
res21: (Int, String) = (1,one)

scala> Tuple2(1,"one")
res26: (Int, String) = (1,one)
```

- Used to group two related items
- Types can be different

Pair

```
scala> val t = (1, "one")
res35: (Int, String) = (1,one)

scala> t._1
res36: Int = 1

scala> t._2
res37: String = one

scala> t.swap
res38: (String, Int) = (one,1)
```

Sequence

```
scala> val s = Seq(1, 2, 3)
s: Seq[Int] = List(1, 2, 3)

scala> s(0)
res31: Int = 1

scala> s.head
res34: Int = 1

scala> s.tail
res35: Seq[Int] = List(2, 3)
```

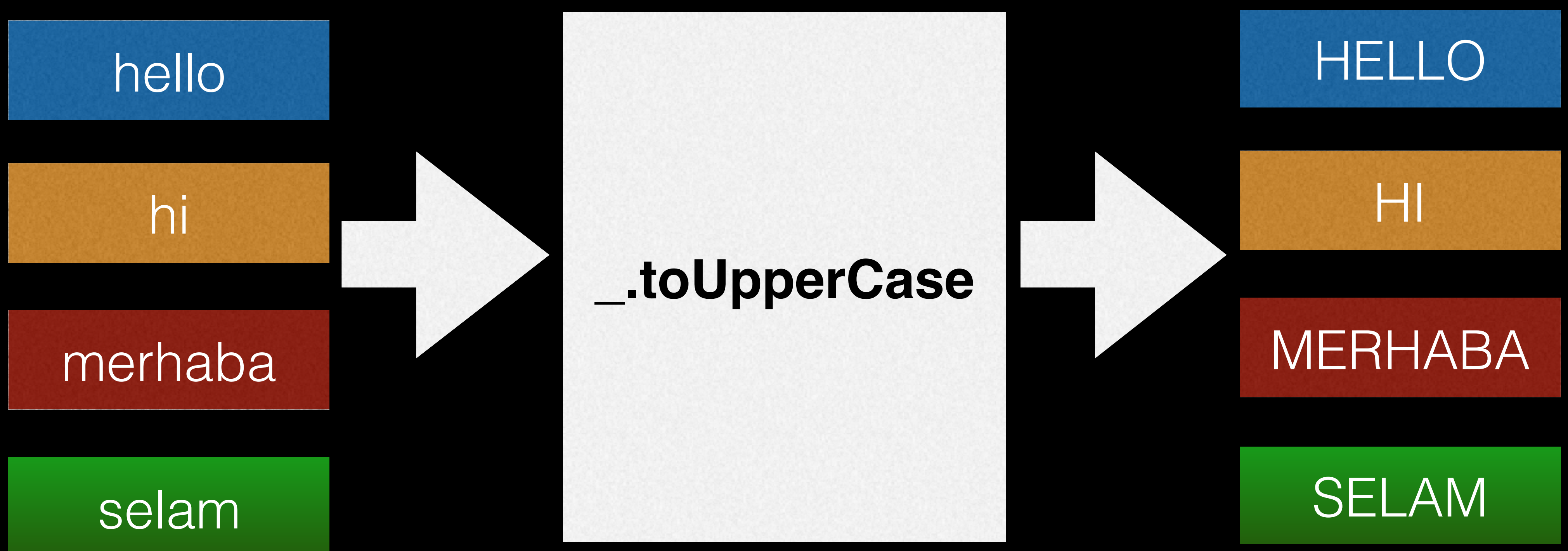
RDD Operations

- RDDs support two types of operations
  - transformations (ex: map, filter)
    Returns new RDD
  - actions (ex: reduce, collect)
    Returns normal values

Transformations

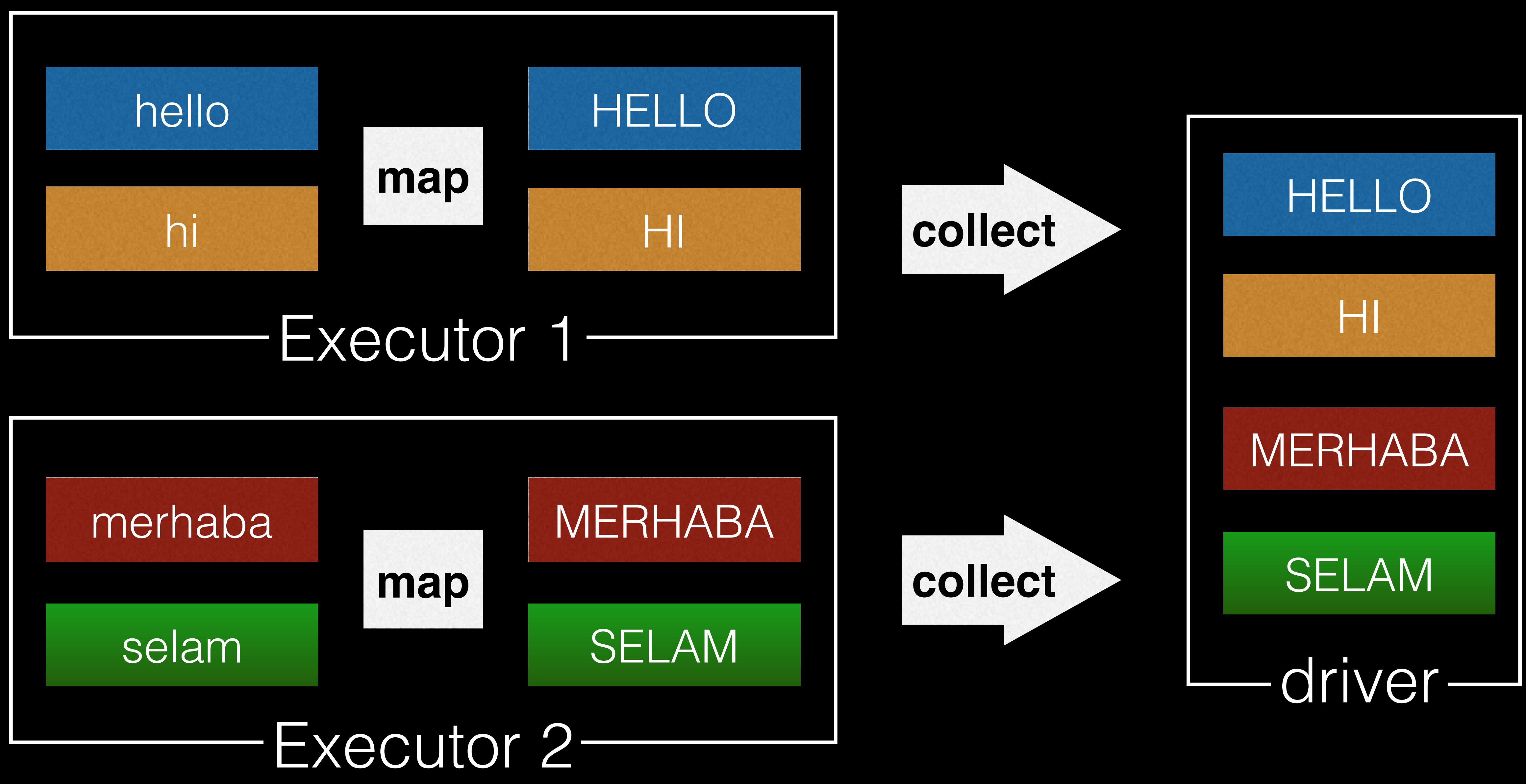
- map
- flatMap
- filter
- distinct

map



- Output RDD has same length as input RDD
- Function is applied to every element, and produces exactly one element

map

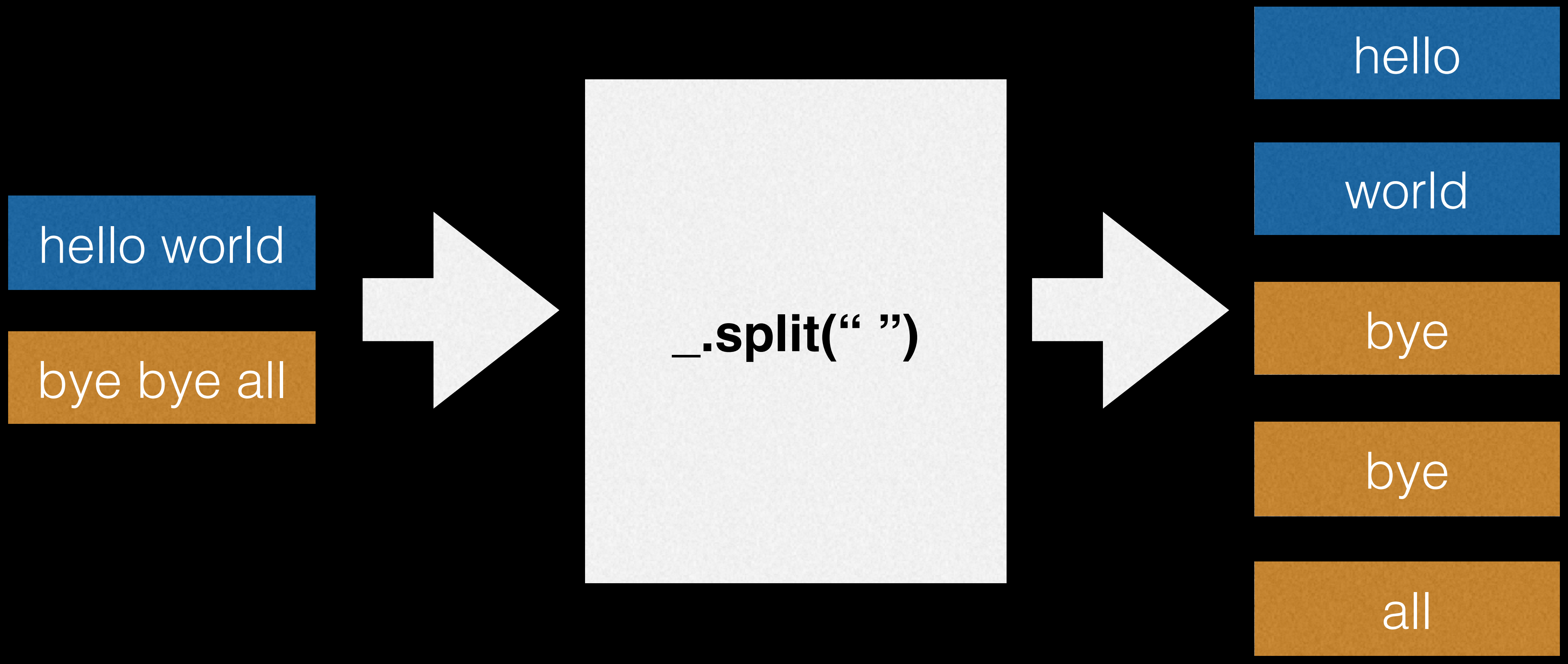

hello
hi
map
HELLO
HI
Executor 1
merhaba
selam
map
MERHABA
SELAM
Executor 2
collect
collect
HELLO
HI
MERHABA
SELAM
driver

map

```
val words =
  sc.parallelize(Seq("hello", "hi", "merhaba", "selam"))

words.map(_.toUpperCase).collect
```

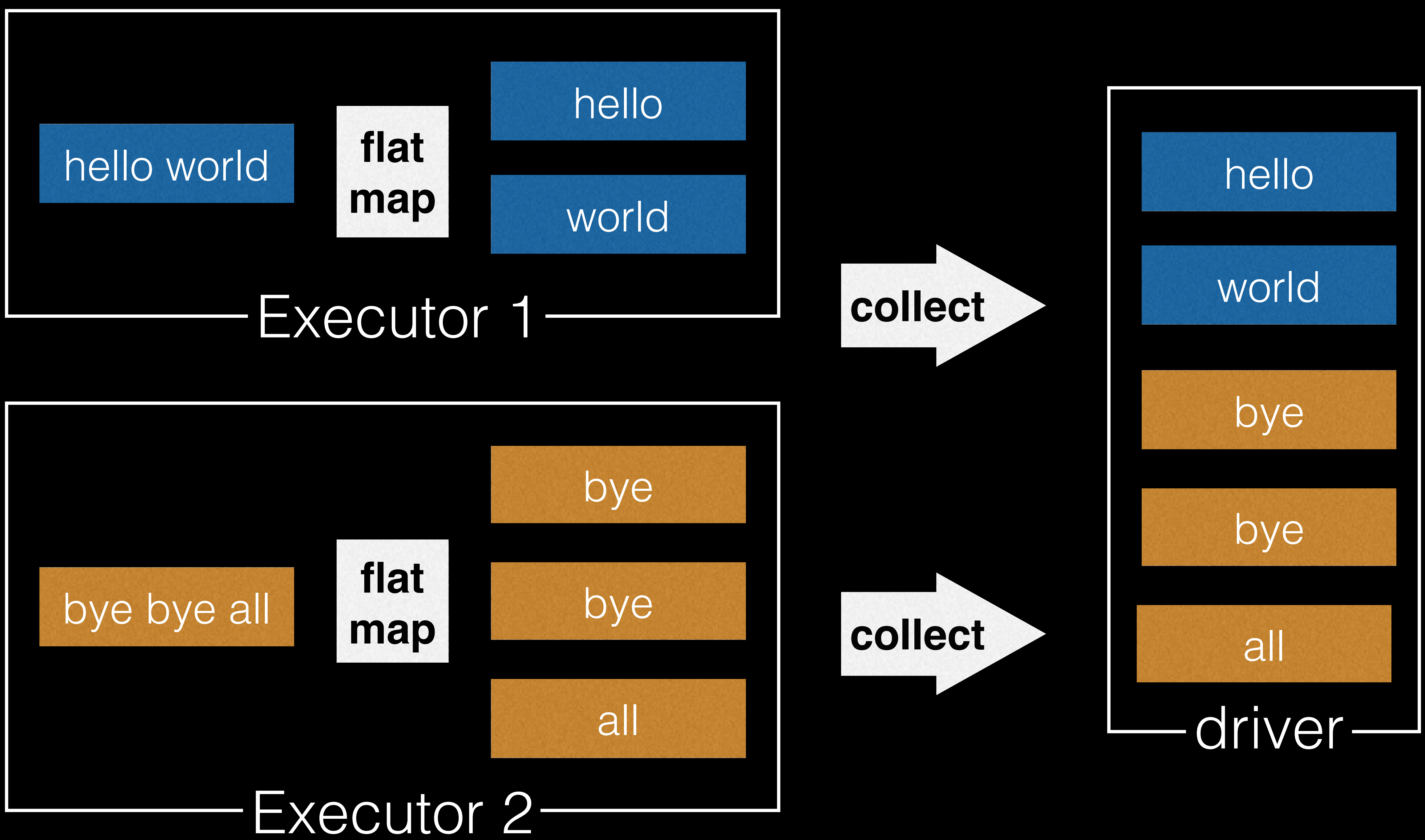
res17: Array[String] = Array(HELLO, HI, MERHABA, SELAM)

flatMap



- Function produces zero or more elements

flatMap


hello world
flat map
hello
world
Executor 1
bye bye all
flat map
bye
bye
all
Executor 2
collect
collect
hello
world
bye
bye
all
driver

flatMap

```
val words =
  sc.parallelize(Seq(“hello world", “bye bye all"))

words.flatMap(_.split(“ ”)).collect
```

res17: Array[String] = Array(hello, world, bye, bye, all)

flatMap

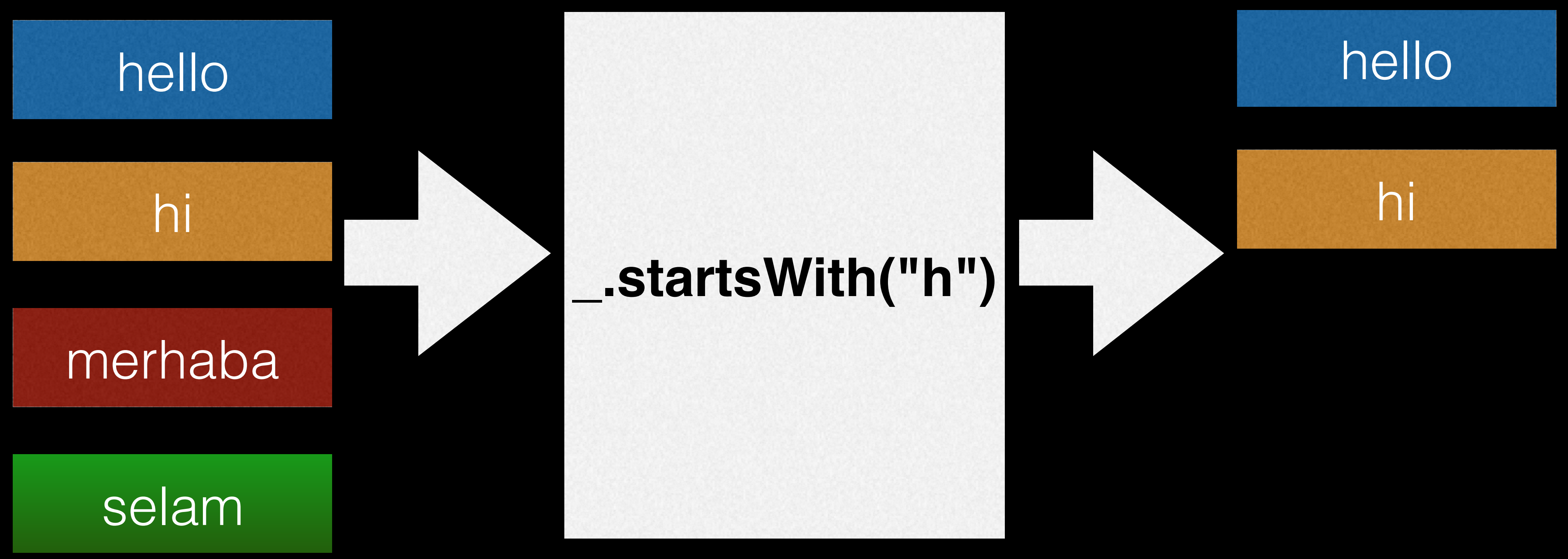
Only Space

```
val words =
  sc.parallelize(Seq(“hello world", “bye bye all”, “ ”))

words.flatMap(_.split(“ ”)).collect
```

res17: Array[String] = Array(hello, world, bye, bye, all)

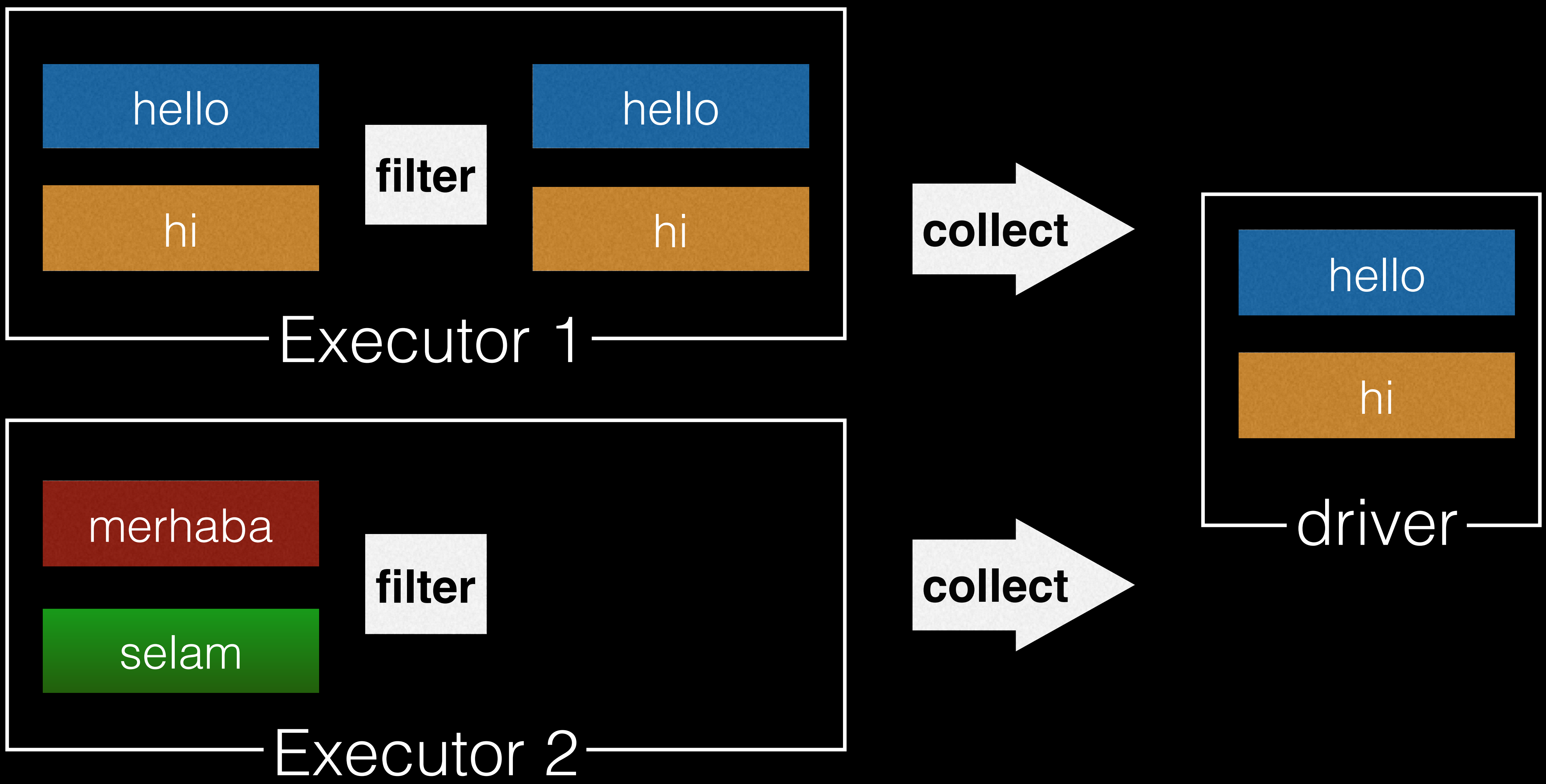
Last element in input didn’t produce any output

filter



- Output RDD’ length is less or equal to input’s
- Functions return boolean
- Only elements who match are produced in output RDD

filter

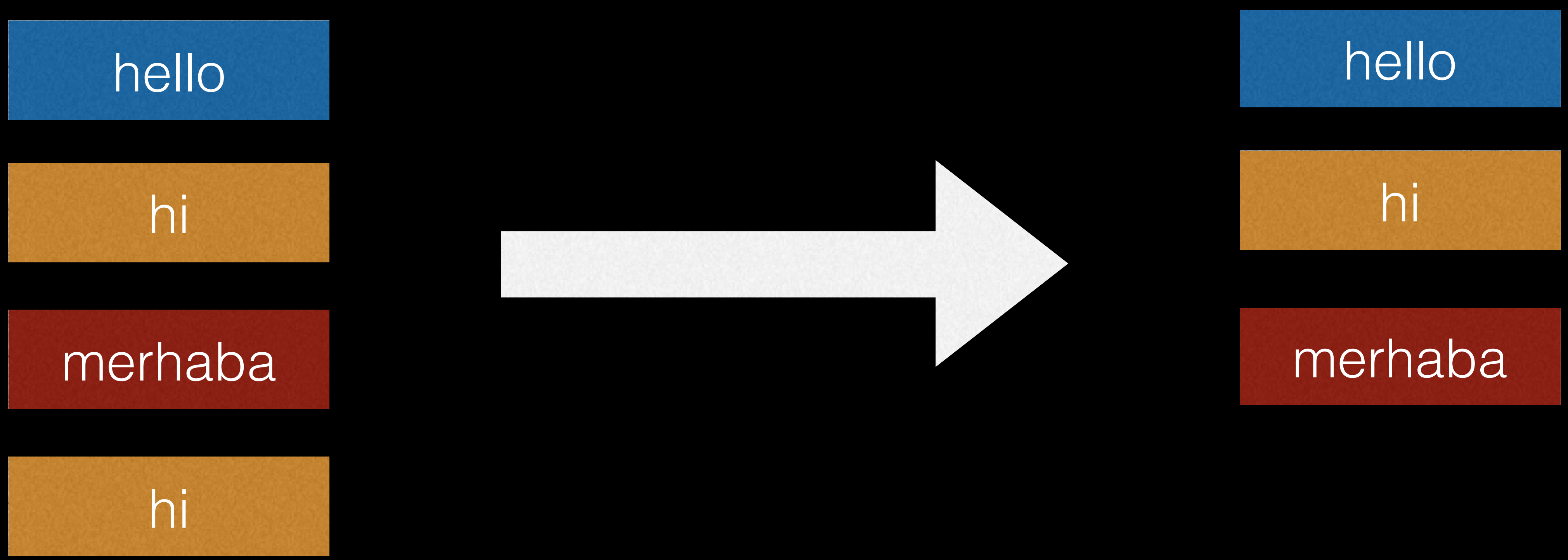

hello
hi
filter
hello
hi
Executor 1
merhaba
selam
filter
Executor 2
collect
collect
hello
hi
driver

filter

```
val words =
  sc.parallelize(Seq("hello", "hi", "merhaba", "selam"))

words.map(_.startsWith(“h”)).collect
```

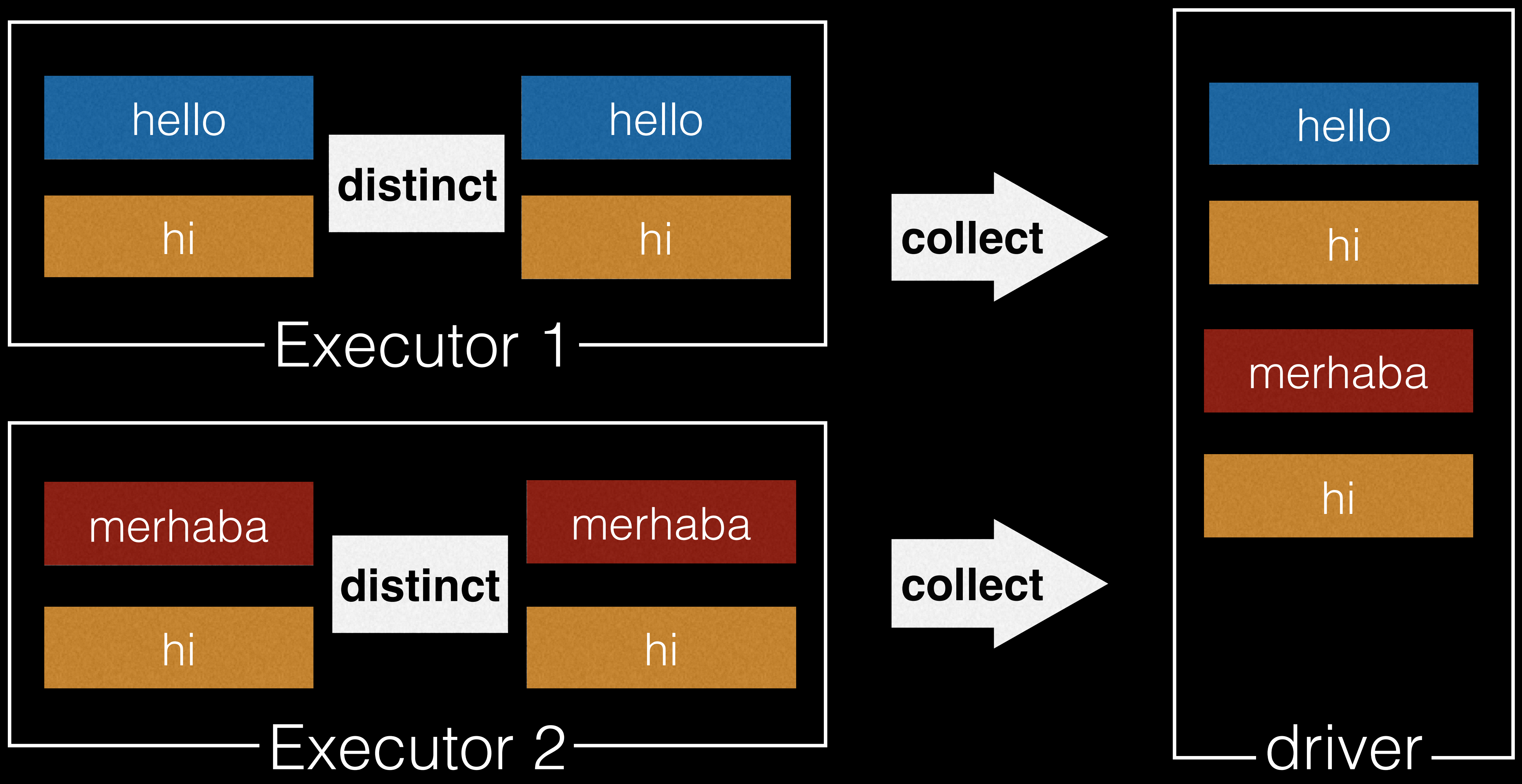
res17: Array[String] = Array(hello, hi)

distinct



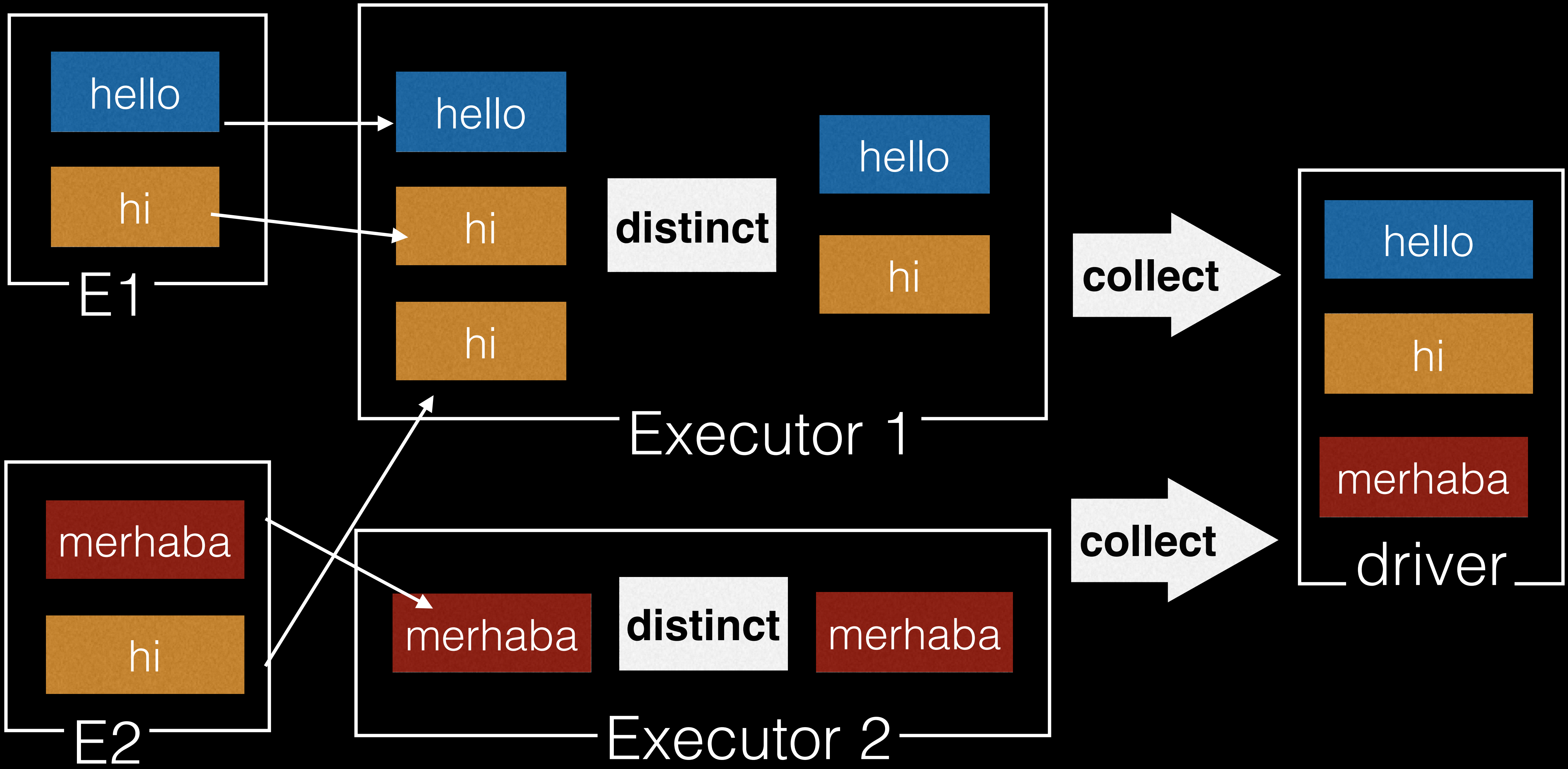
- Output RDD' length is less or equal to input's
- Functions return boolean
- Only elements who match are produced in output RDD

distinct



Executors are independent, we need more work!

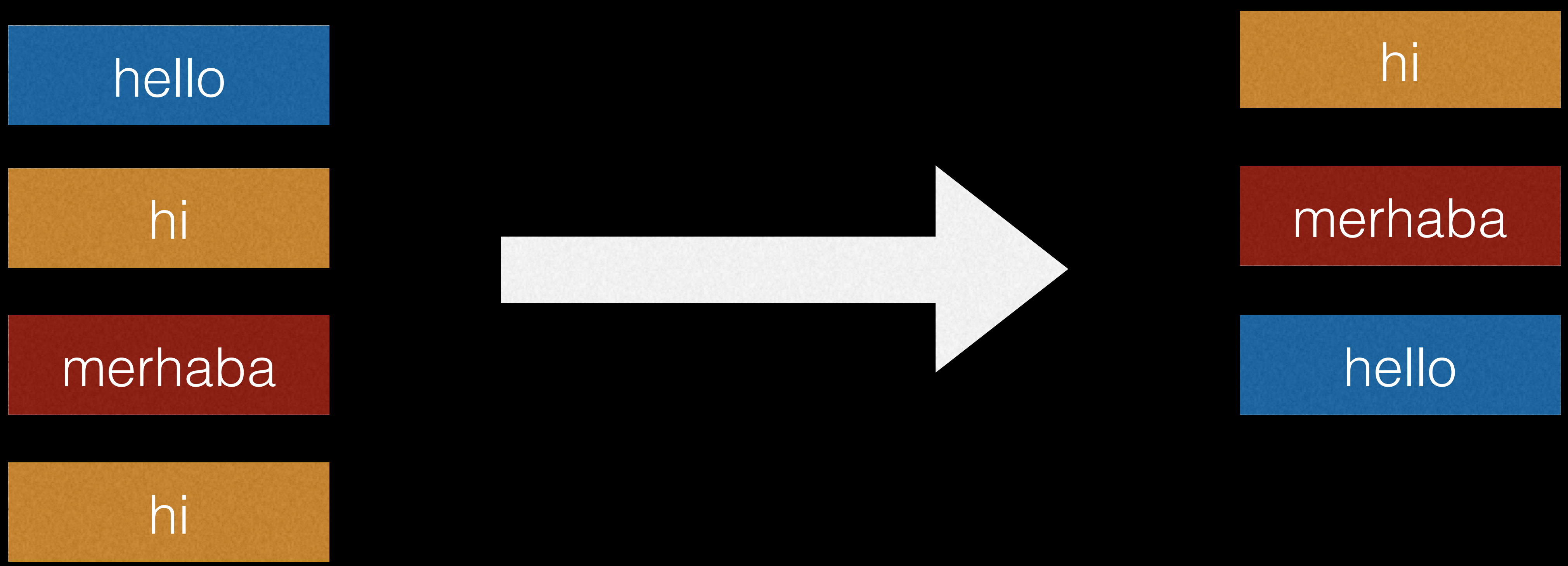
shuffle operation (repartitioning of data)



Shuffle operation

- re-distributing data so that it’s grouped differently across partitions
- repartitioning data is expensive operation
- moves data across network, which can be slow
- however, sometimes it is necessary

distinct



- Usually the order of RDD is lost (elements are shuffled)

distinct

```
val words =
  sc.parallelize(Seq("hello", "hi", "merhaba", "hi"))

words.distinct.collect
```

res17: Array[String] = Array(hello, hi, merhaba)