CORK INSTITUTE OF TECHNOLOGY
INSTITIÚID TEICNEOLAÍOCHTA CHORCAÍ

Big Data Processing

L22-23: Anatomy of the Execution of a Spark Core Program

Dr. Ignacio Castineiras
Department of Computer Science

Outline

1. Setting the Context.
2. RDD Private Side: Partitions and Lineage.
3. Spark Application: Jobs, Stages and Tasks.

Outline

1. Setting the Context.
2. RDD Private Side: Partitions and Lineage.
3. Spark Application: Jobs, Stages and Tasks.

Setting the Context

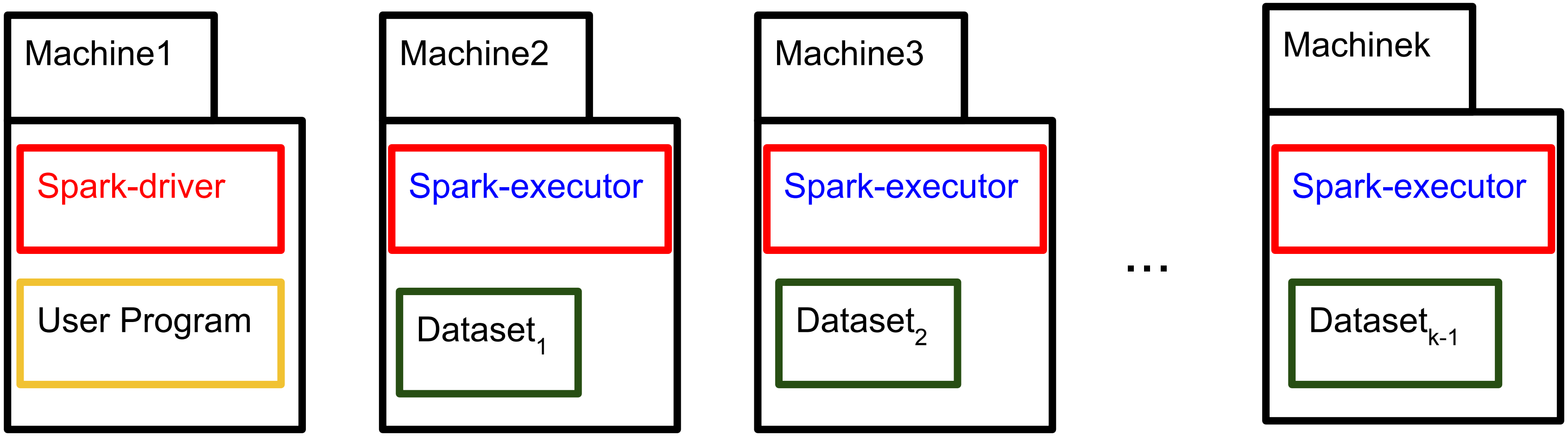
Coming back to the point in which we only had studied Spark Core…

Setting the Context

Let’s put together all the ingredients
we knew by then…

Setting the Context

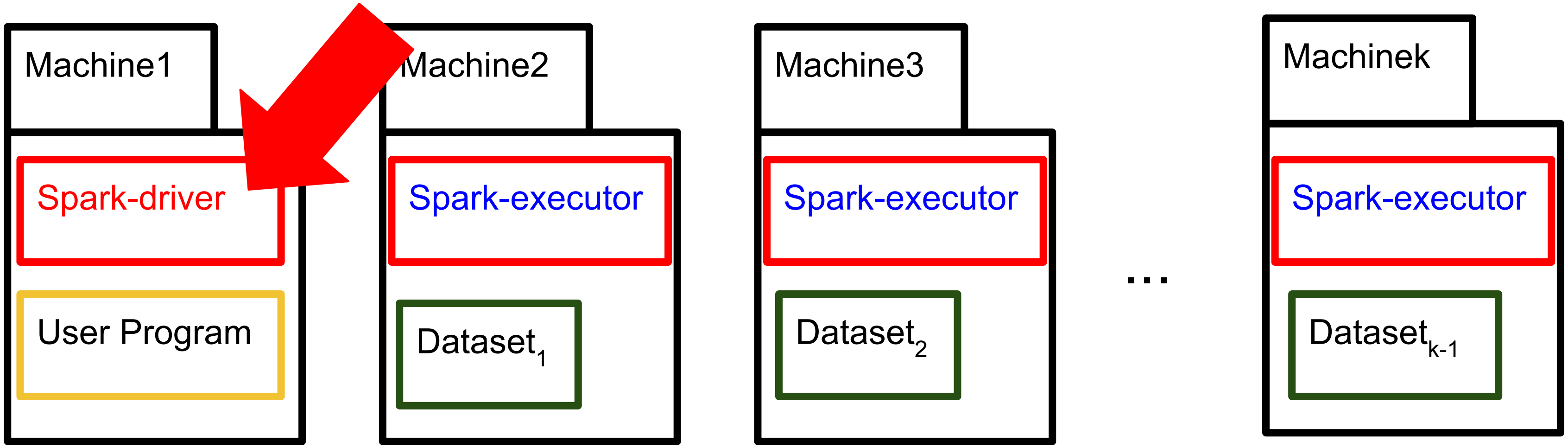
1. We have a **cluster of computers**, connected among them so as to support the distributed computation.



Setting the Context

2. One machine contains the Spark user program.

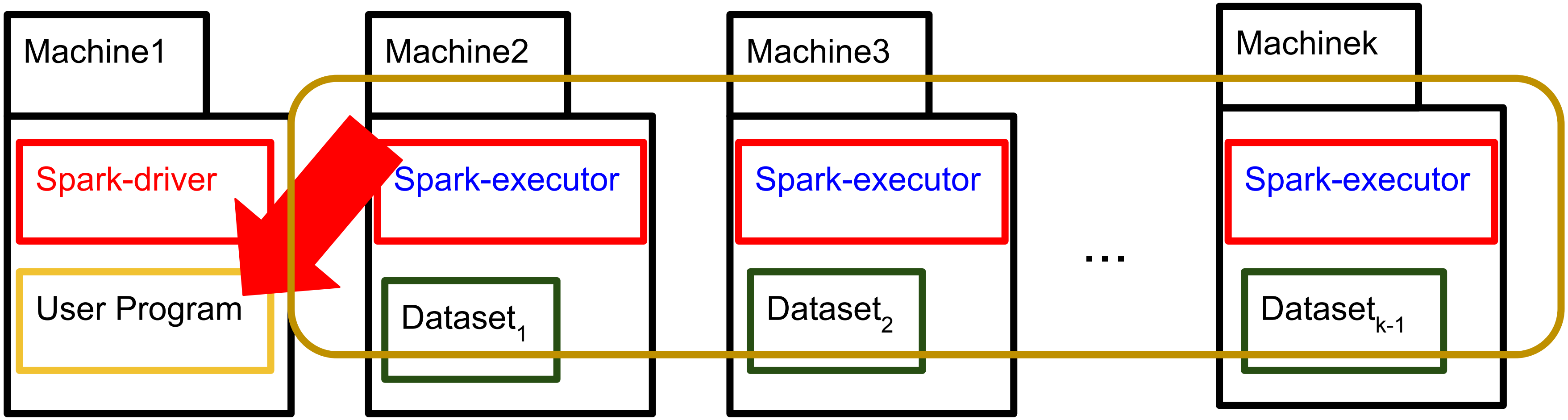
   This machine -more specifically, one CPU core of the machine- runs the **Spark driver process (master)** by executing the main( ) method of the program .



Setting the Context

3. We know by now the Spark user program is based on the RDD public API. It has the following life-cycle:

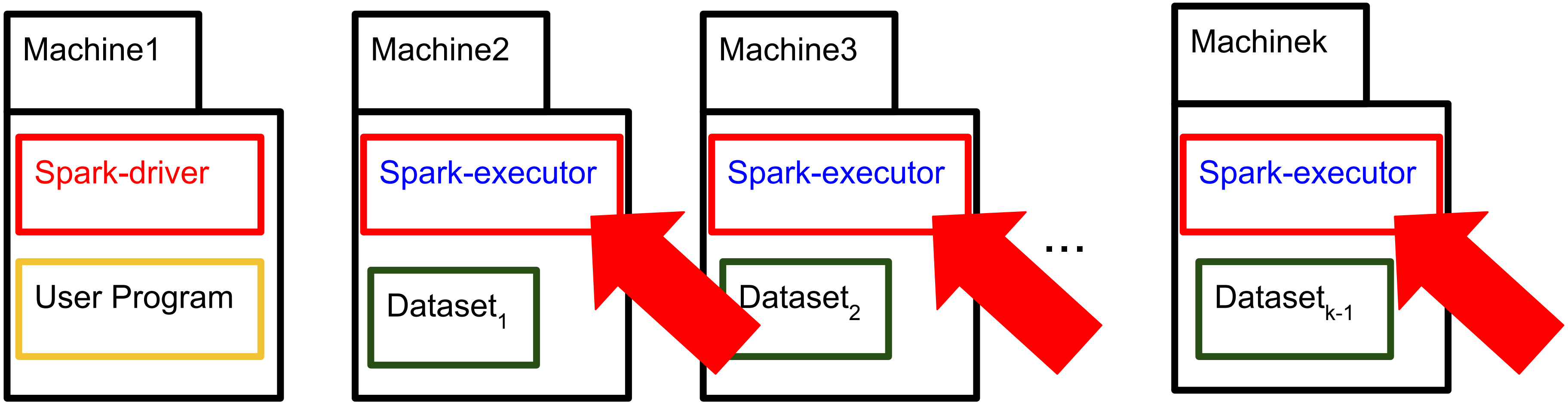
   a. Create some input RDDs from external data.
   b. Transform them to define new RDDs using transformations.
   c. Persist any intermediate RDDs that will need to be reused.
   d. Launch actions to kick off a distributed computation.



Setting the Context

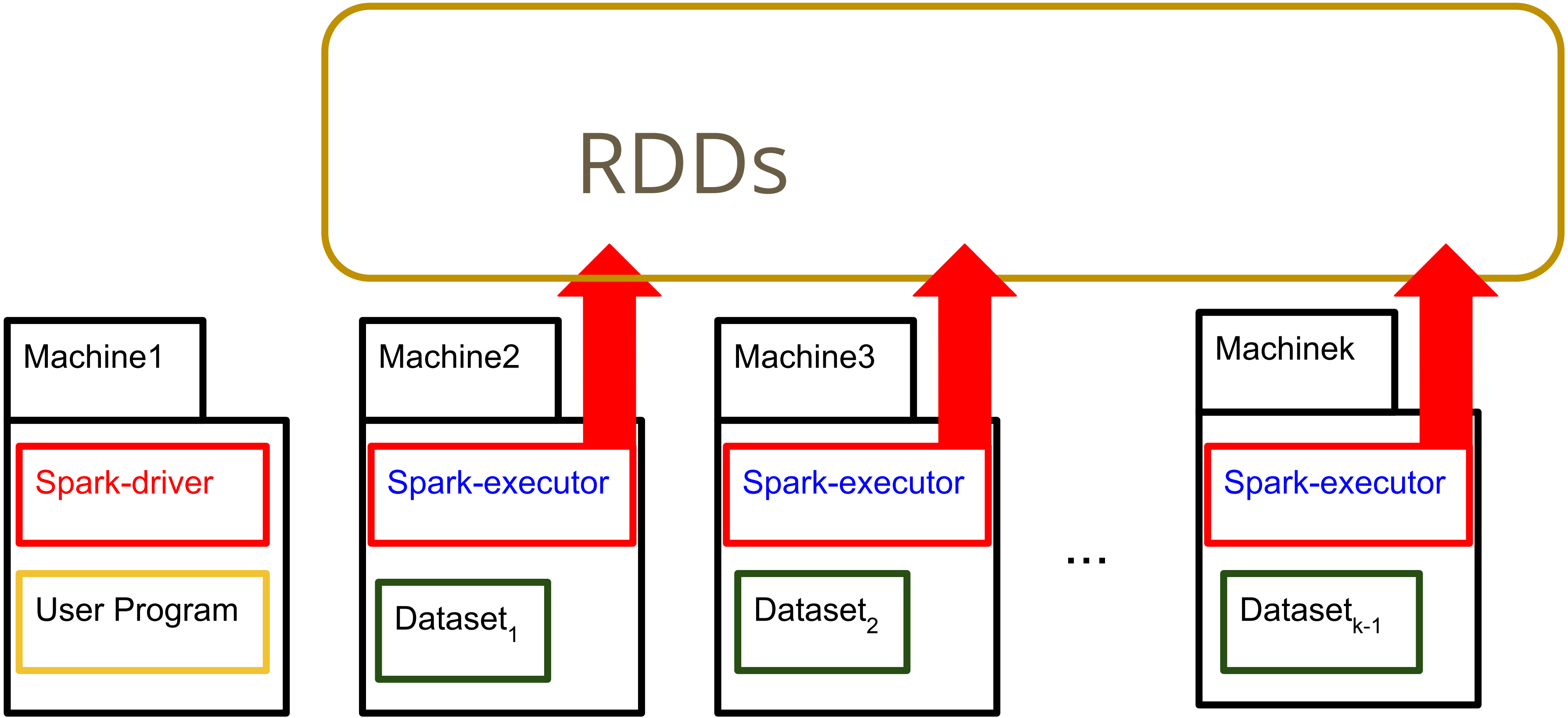
4. We know these RDD operations-based program is performed by the **Spark executor processes (slaves)** of the remaining machines, which use:
   a. Their CPU for computing such RDDs.
   b. Their memory to store such RDDs.

a. Create some input RDDs from external data.
b. Transform them to define new RDDs using transformations.
c. Persist any intermediate RDDs that will need to be reused.
d. Launch actions to kick off a distributed computation.



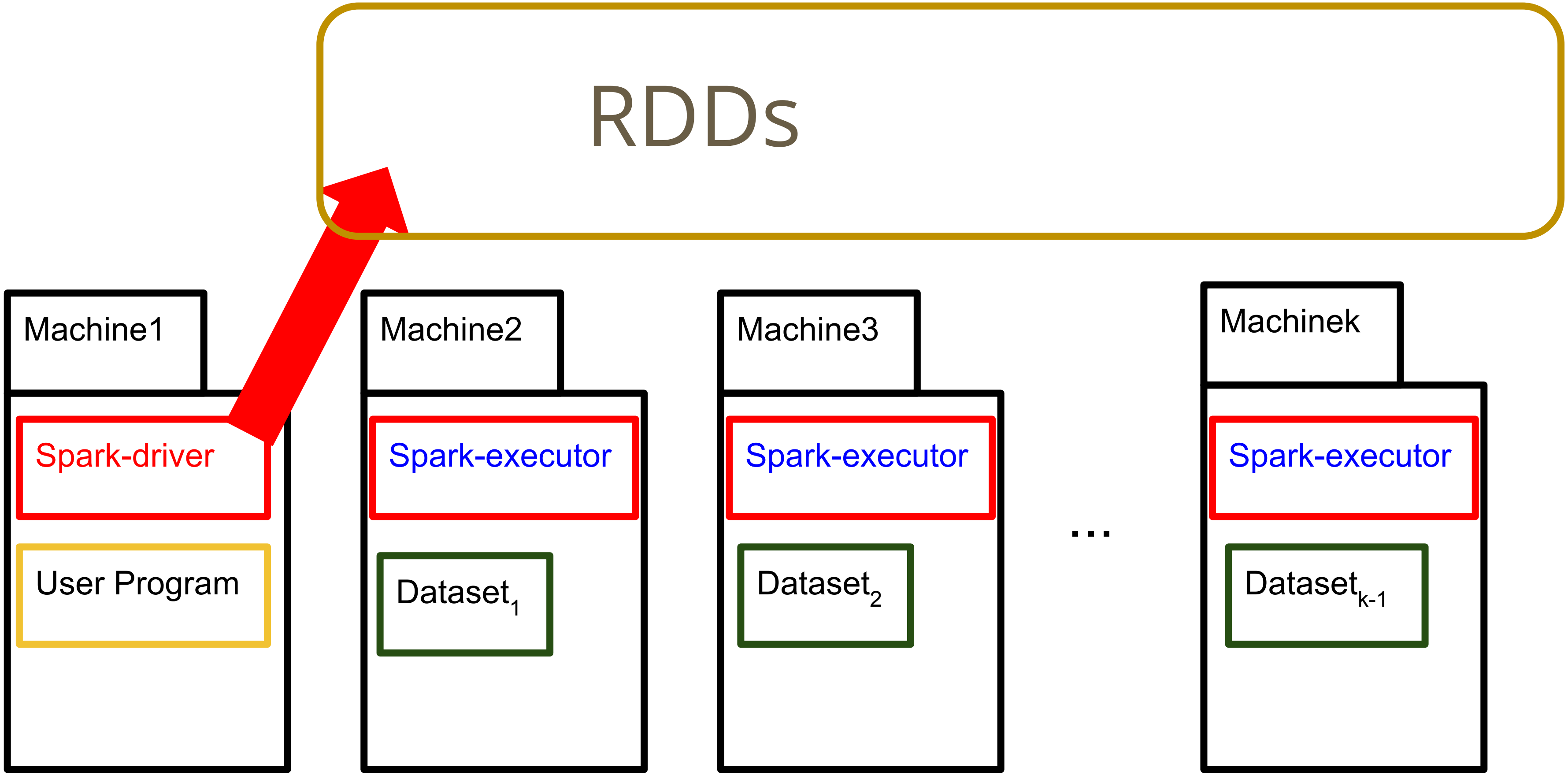
Setting the Context

4. We know these RDD operations-based program will be performed by the **Spark executor processes (slaves)** of the remaining machines, which use:
   a. Their CPU for computing such RDDs.
   b. Their memory to store such RDDs.



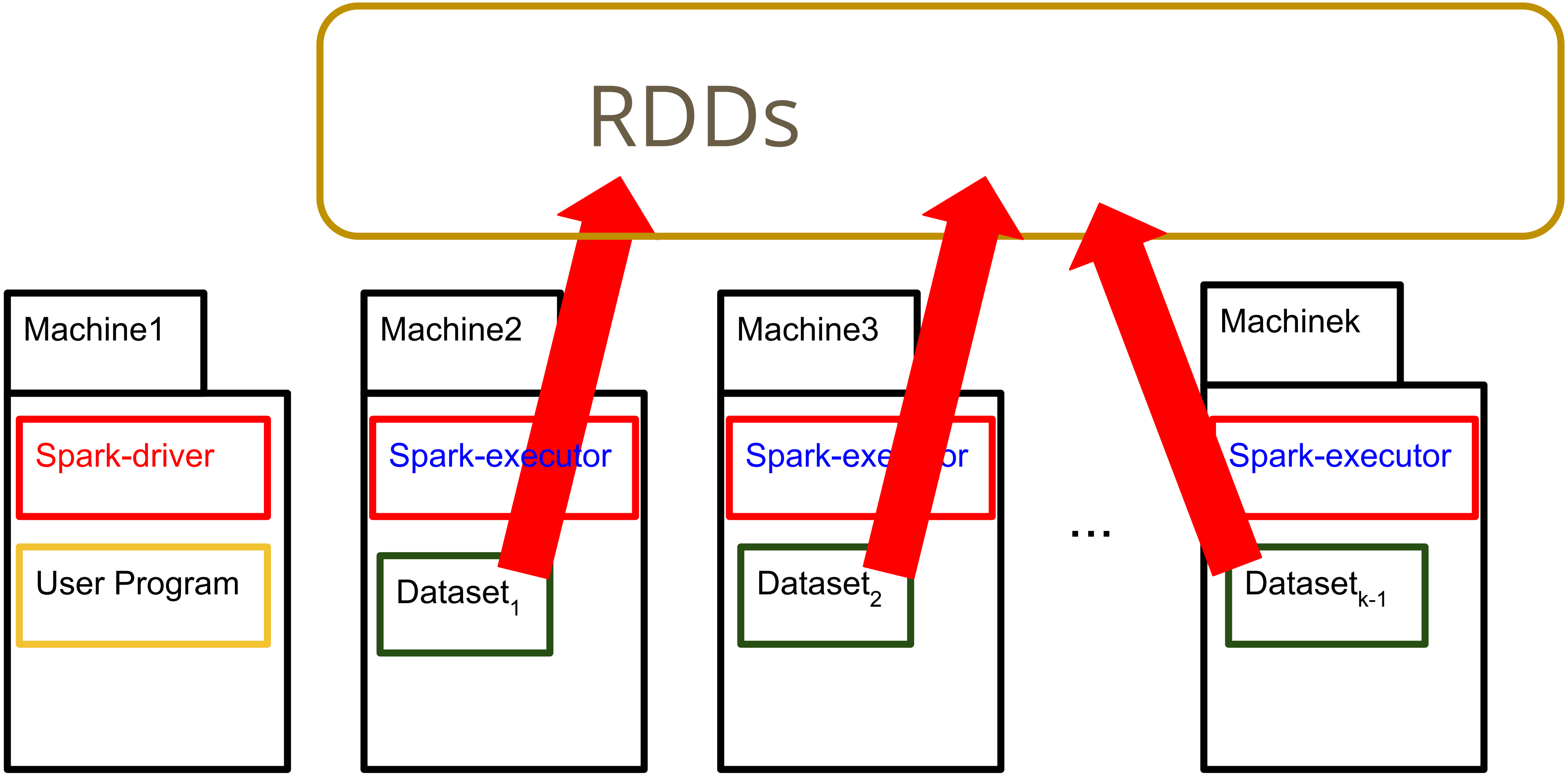
Setting the Context

5. We know the **creation** operations create an RDD by:
   a. Parallelising a List from the driver.



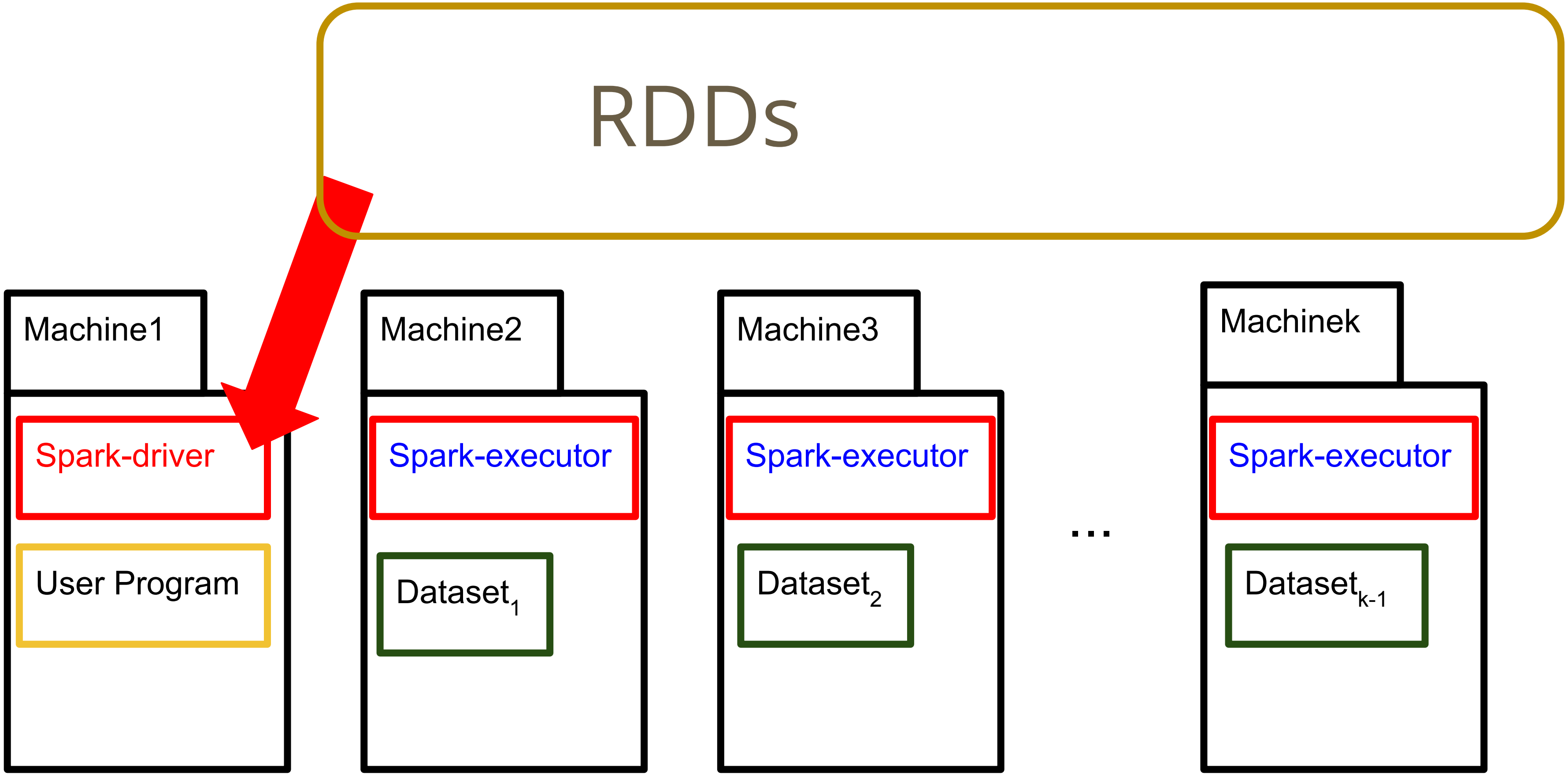
Setting the Context

5. We know the **creation** operations create an RDD by:
   a. Parallelising a List from the driver.
   b. Loading the textFile content from a dataset.



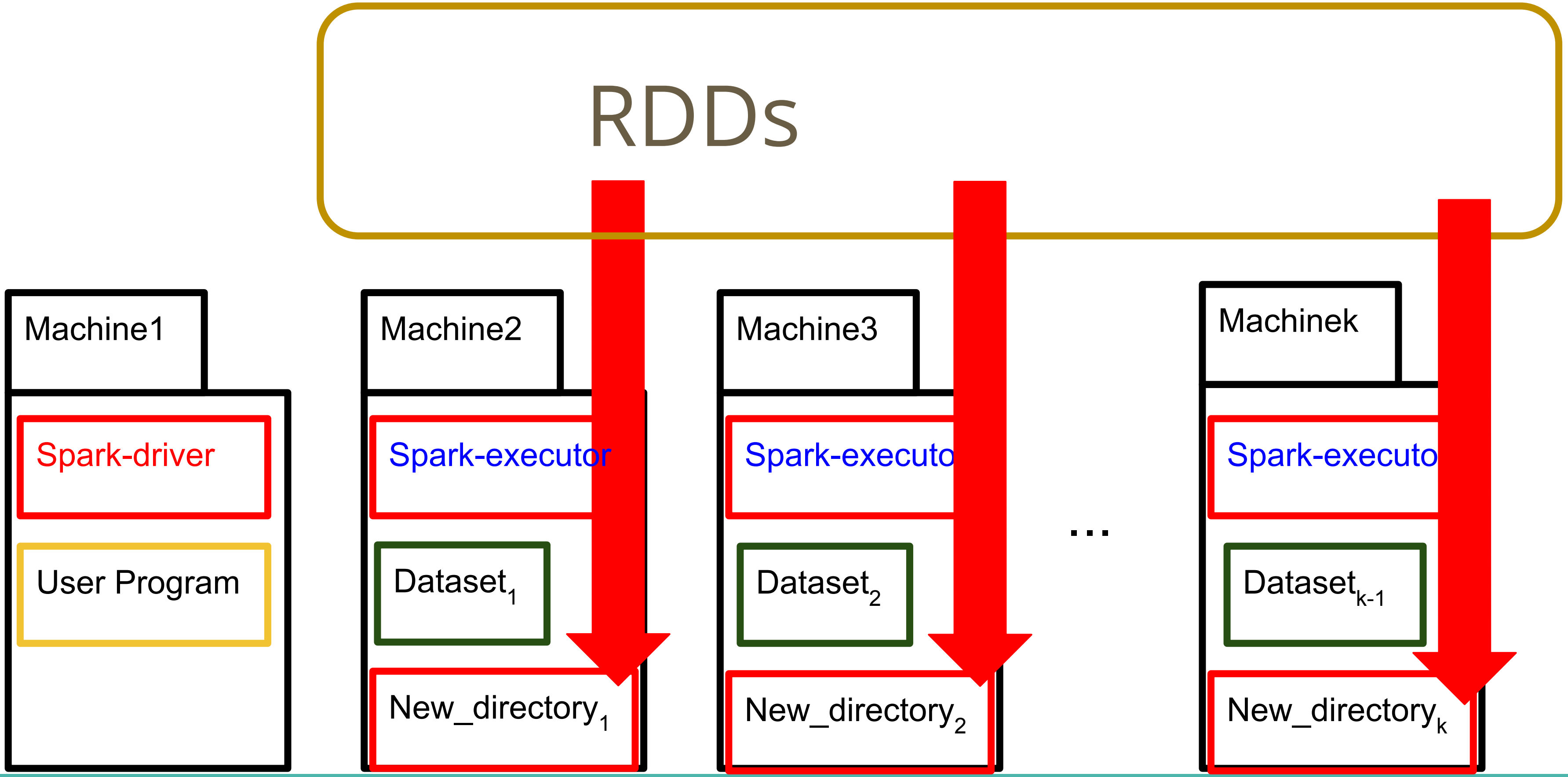
Setting the Context

6. We know the **action** operations produce a result by:
    a. Returning some info to the driver (for it to be printed by the screen).



Setting the Context

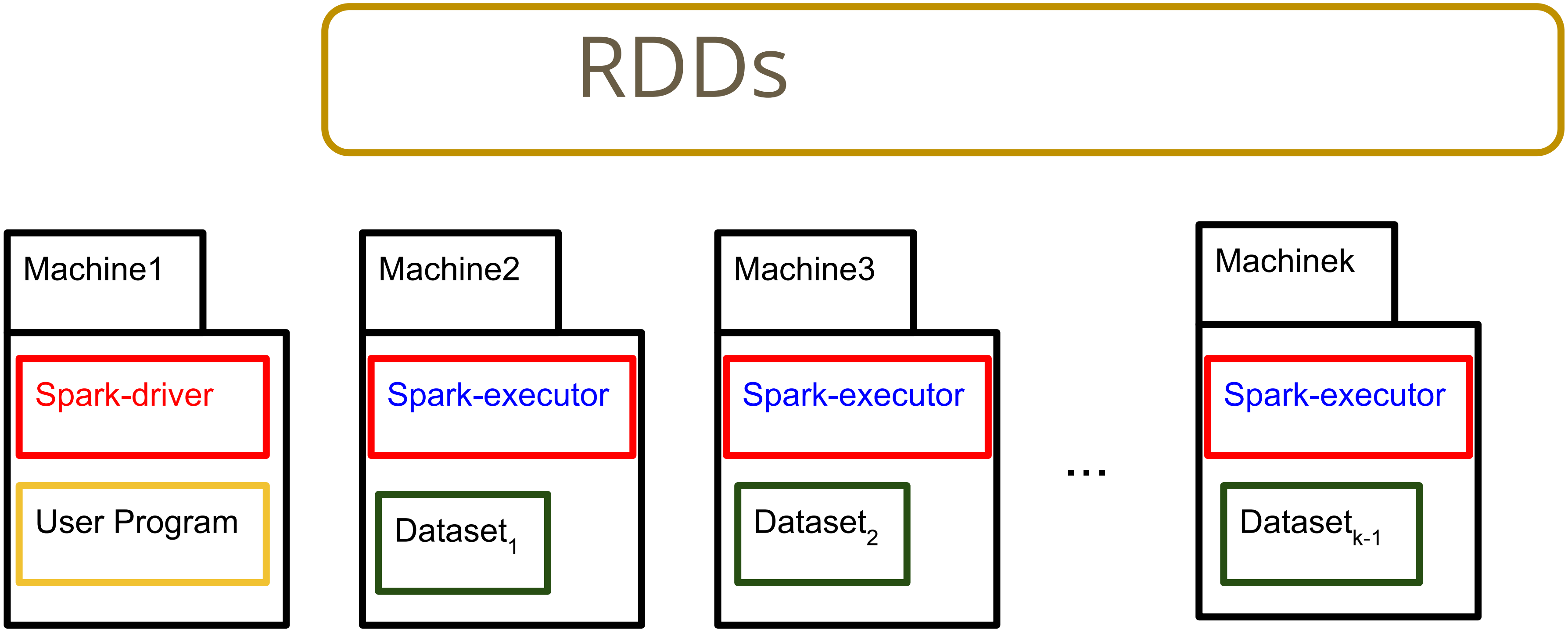
6. We know the **action** operations produce a result by:
   a. Returning some info to the driver (for it to be printed by the screen).
   b. Storing an RDD into a new directory.



Setting the Context

But we still don’t know:

- How RDDs are internally represented (the ADT private side).
- How the Spark-executors operate to compute these RDDs.

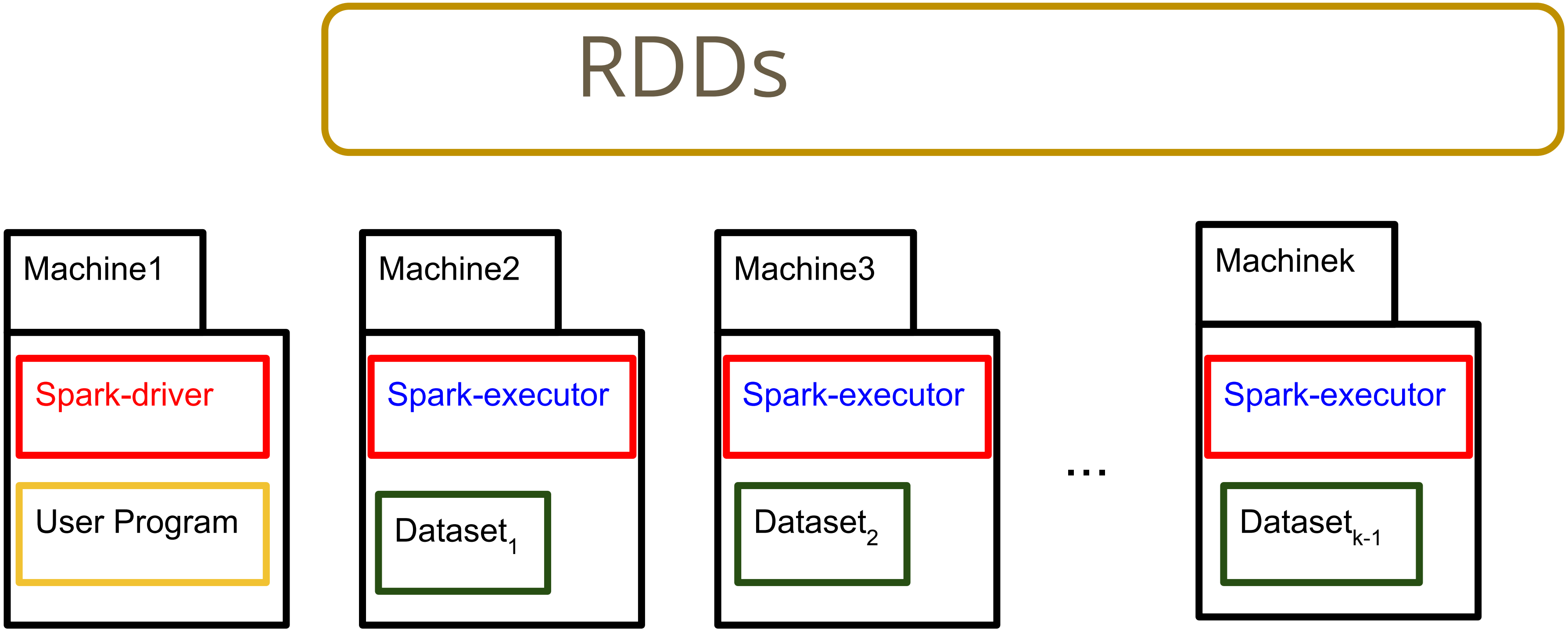


Setting the Context

But we still don't know:

- How RDDs are internally represented (the ADT private side).
- How the Spark-executors operate to compute these RDDs.

Understanding this is our goal for today!



Outline

1. Setting the Context.
2. RDD Private Side: Partitions and Lineage.
3. Spark Application: Jobs, Stages and Tasks.

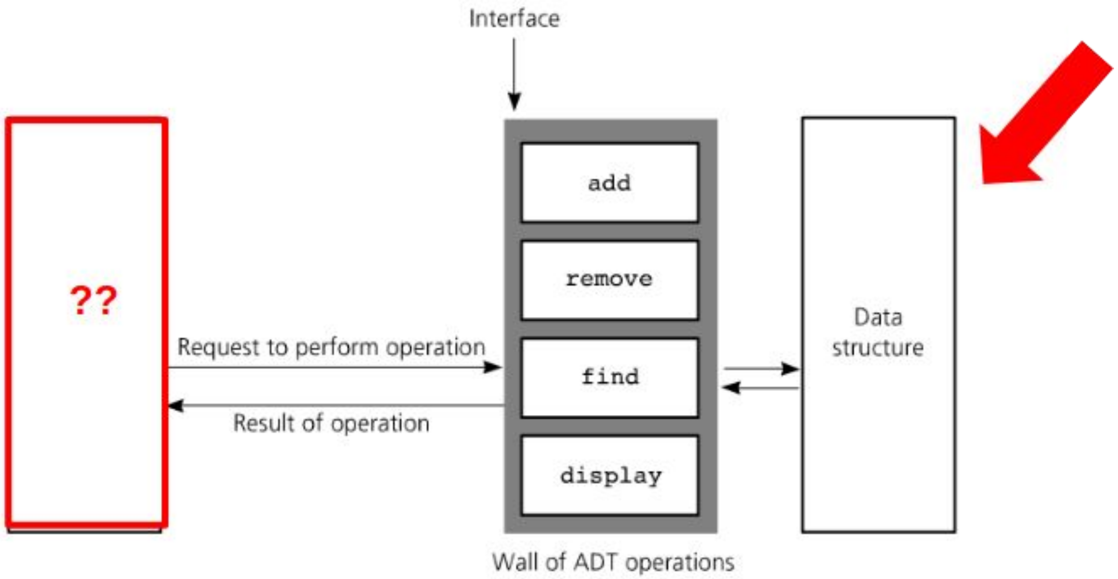
Outline

1. Setting the Context.
2. RDD Private Side: Partitions and Lineage.
    a. Internal Representation.
    b. Partitions.
    c. Lineage: Narrow and Wide Transformations.
    d. Lineage: Lazy evaluation.
    e. Lineage: Fault tolerant.
3. Spark Application: Jobs, Stages and Tasks.

Internal Representation

- The **ADT private side** puts on the feet of the data developer.
  To do so, it has to sort out another 2 main questions:

3. **How** is the data internally represented?
   Specify the concrete data structures used to layout the data.
4. **How** is each operation internally implemented?

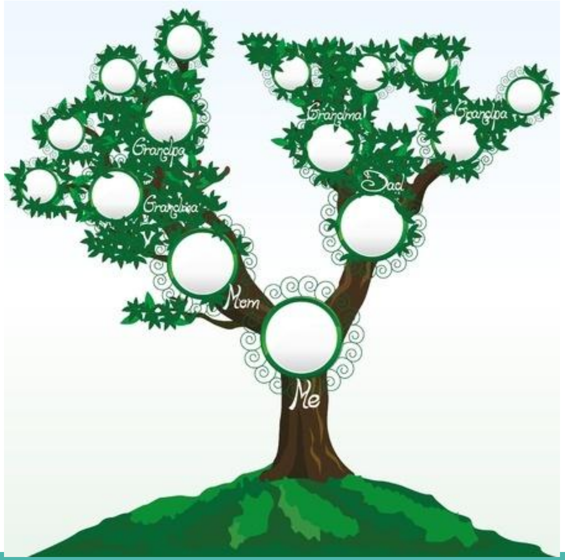


Internal Representation

Let’s go with the ADT private side:

3. **How** is the data internally represented?
   Specify the concrete data structures used to layout the data.

- An RDD is internally represented via...
  1. A set of **partitions**
  2. Enriched with **lineage** metadata for their re-computation.

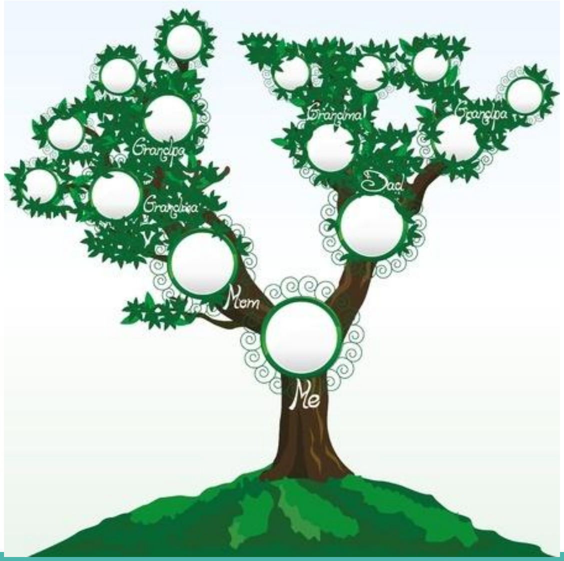


Internal Representation

Let's go with the ADT private side:

3. **How** is the data internally represented?
   Specify the concrete data structures used to layout the data.

- An RDD is internally represented via...
  1. A set of **partitions**
  2. Enriched with **lineage** metadata for their re-computation.



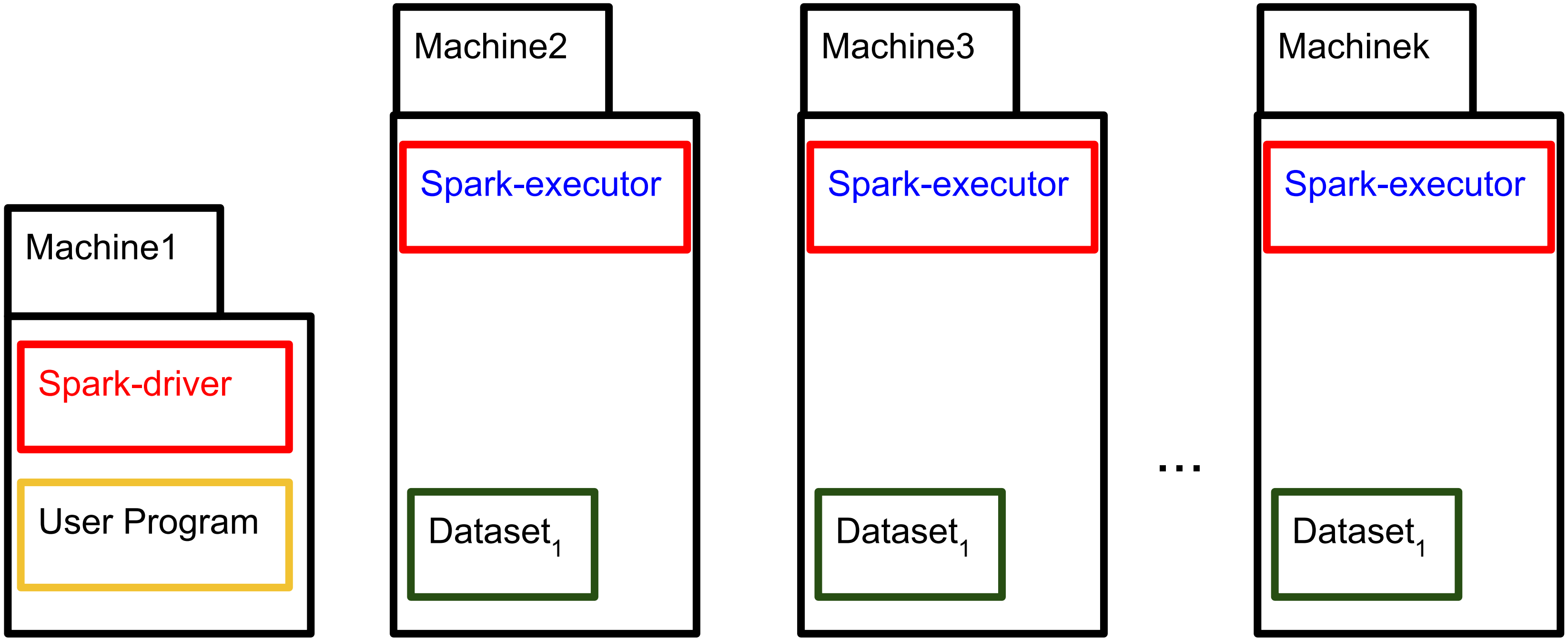
Outline

1. Setting the Context.
2. RDD Private Side: Partitions and Lineage.
    a. Internal Representation.
    b. Partitions.
    c. Lineage: Narrow and Wide Transformations.
    d. Lineage: Lazy evaluation.
    e. Lineage: Fault tolerant.
3. Spark Application: Jobs, Stages and Tasks.

Partitions

The motivation for being partitioned is straightforward:

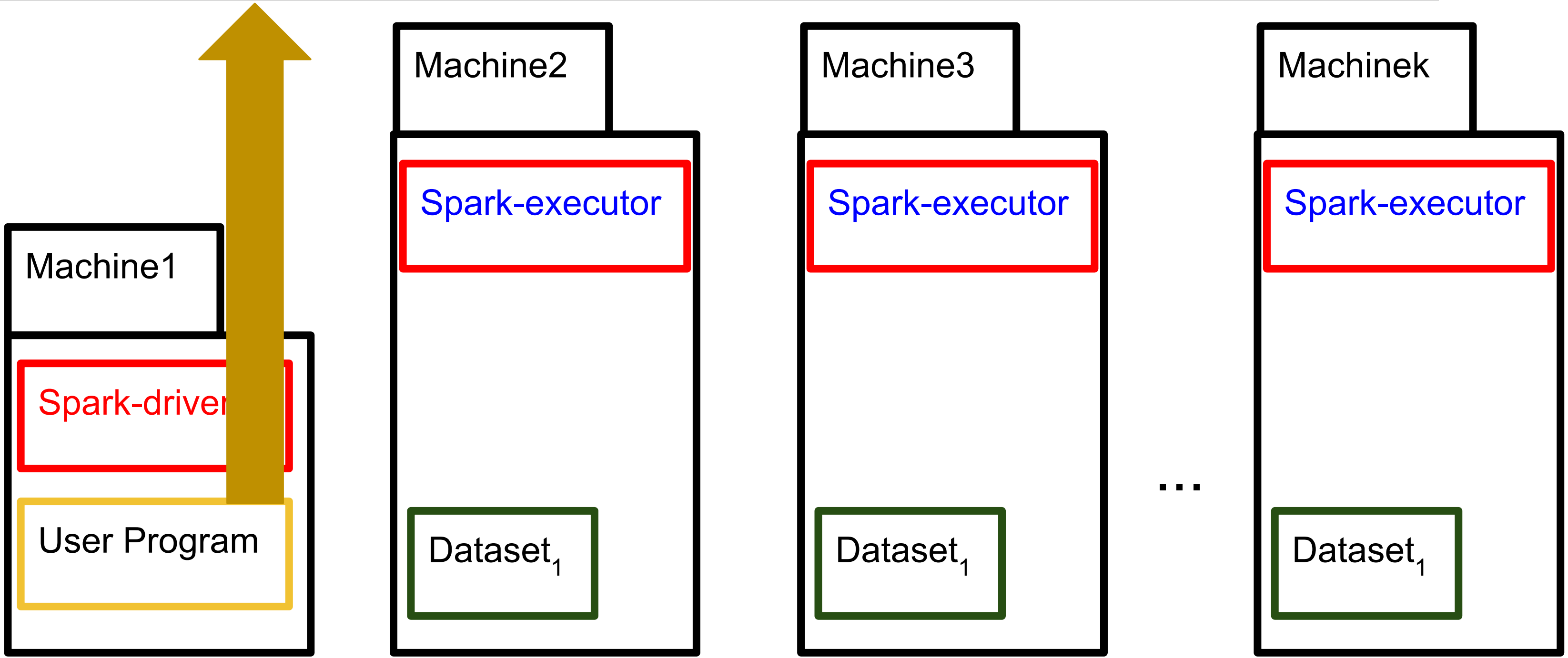
- If we decide to internally implement an RDD as a partitioned data structure, then we can distribute it among the Spark executor processes of the cluster, turning the compute and storage of such RDD into a collaborative task.



Partitions

- ➢ For example, given an inputRDD obtained from parallelizing a list:

```
inputRDD  = sc.parallelize( [ 1, 2, 3, 4, 5, 6, 7 ] )
```



Machine1

Spark-driver

User Program

Machine2

Spark-executor

Dataset$_1$

Machine3

Spark-executor

Dataset$_1$

...

Machinek

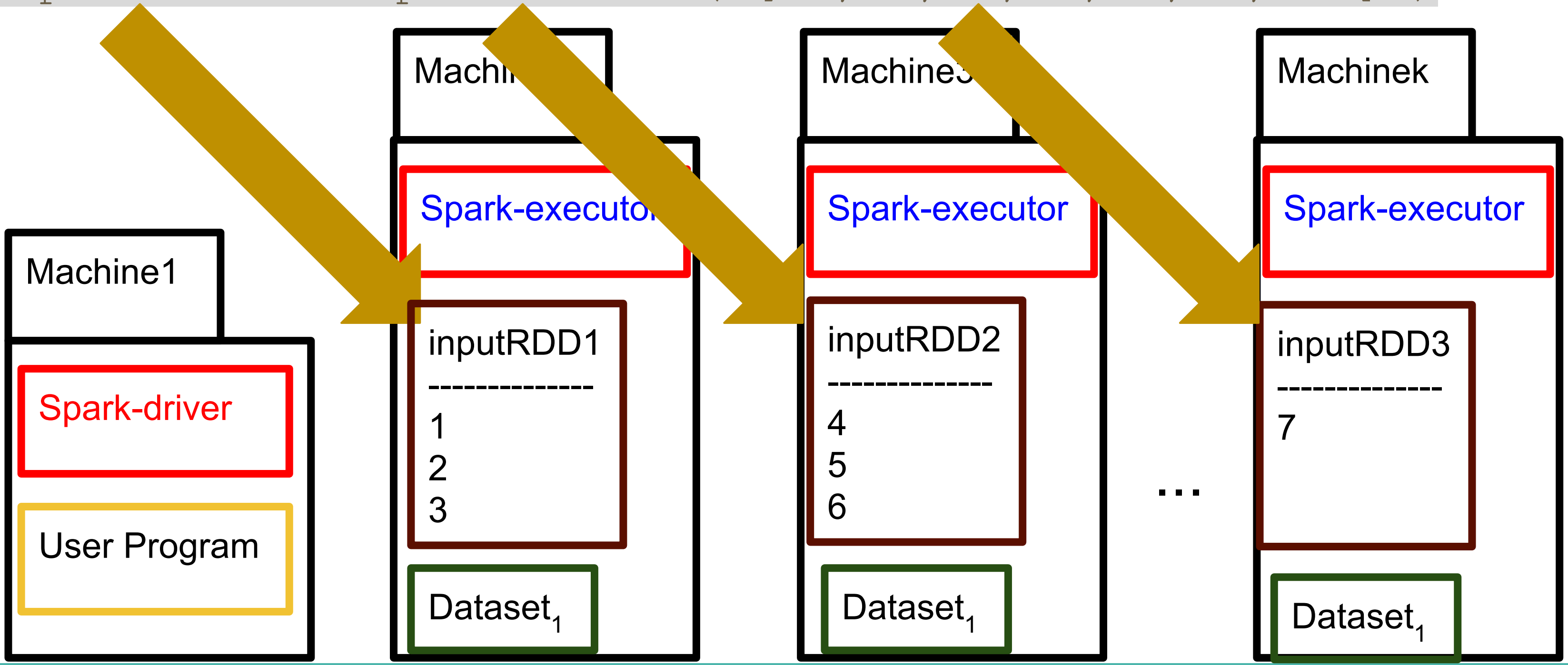
Spark-executor

Dataset$_1$

Partitions

➢ For example, given an inputRDD obtained from parallelizing a list:
We can represent it with 3 partitions, one per Spark executor process.

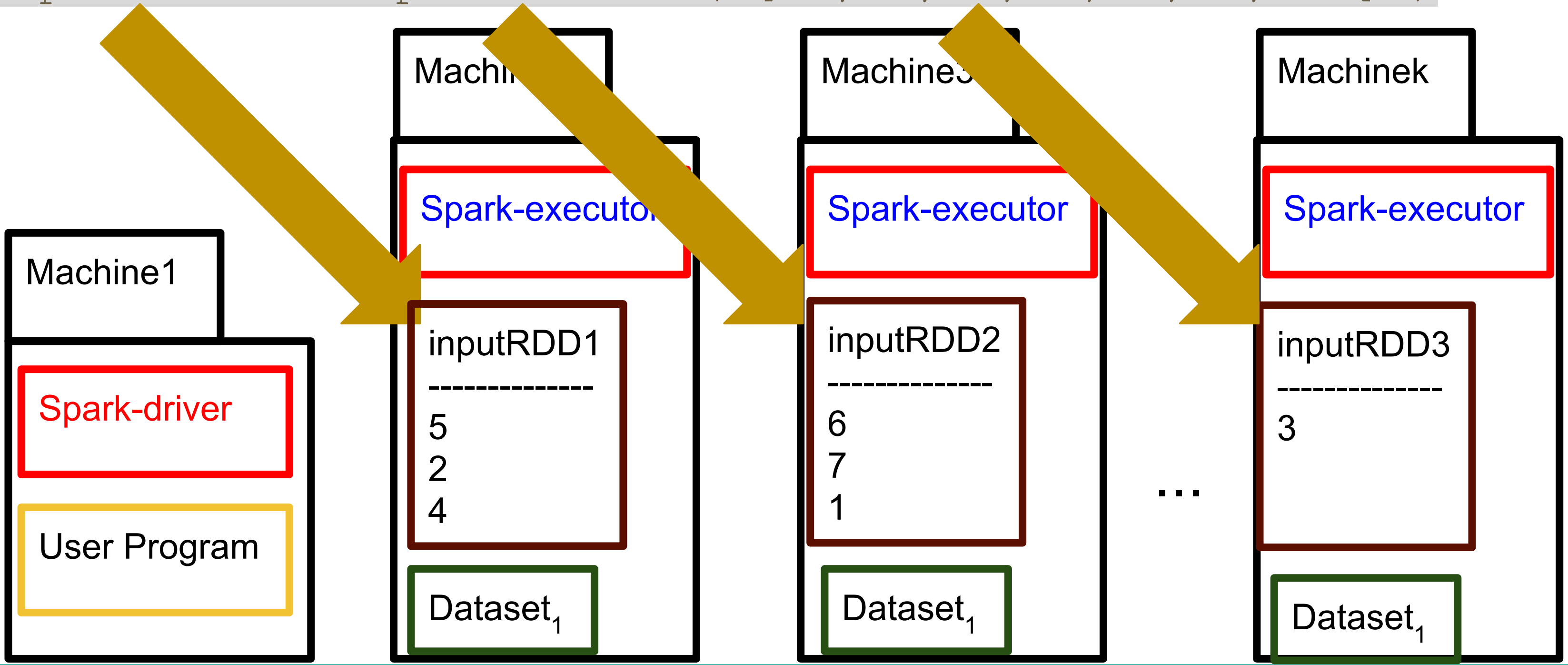
```
inputRDD  = sc.parallelize( [ 1, 2, 3, 4, 5, 6, 7 ] )
```



Partitions

➢ For example, given an inputRDD obtained from parallelizing a list:
Or with this other distribution, as well with 3 partitions, one per Spark executor process.

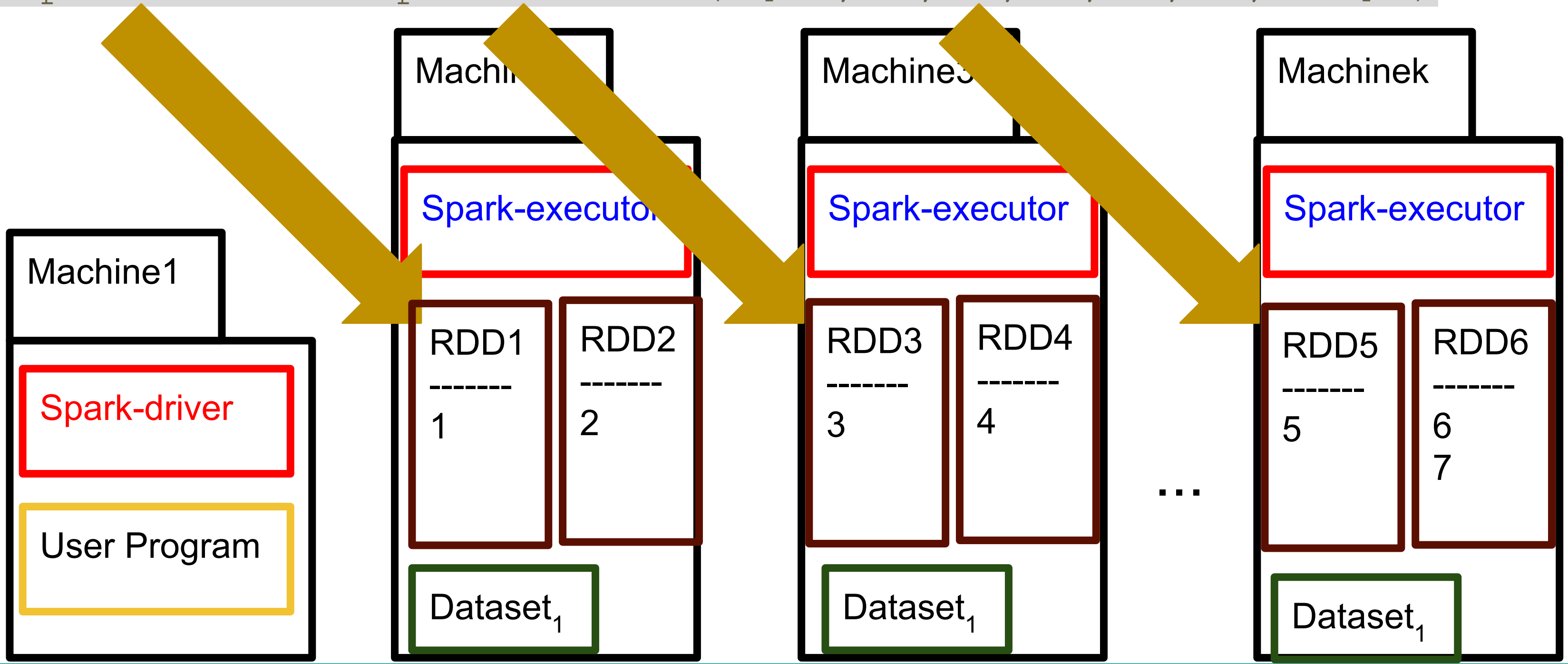
```
inputRDD  = sc.parallelize( [ 1, 2, 3, 4, 5, 6, 7 ] )
```



Partitions

- For example, given an inputRDD obtained from parallelizing a list:
  Or with this other distribution, now with 6 partitions, two per Spark executor process.

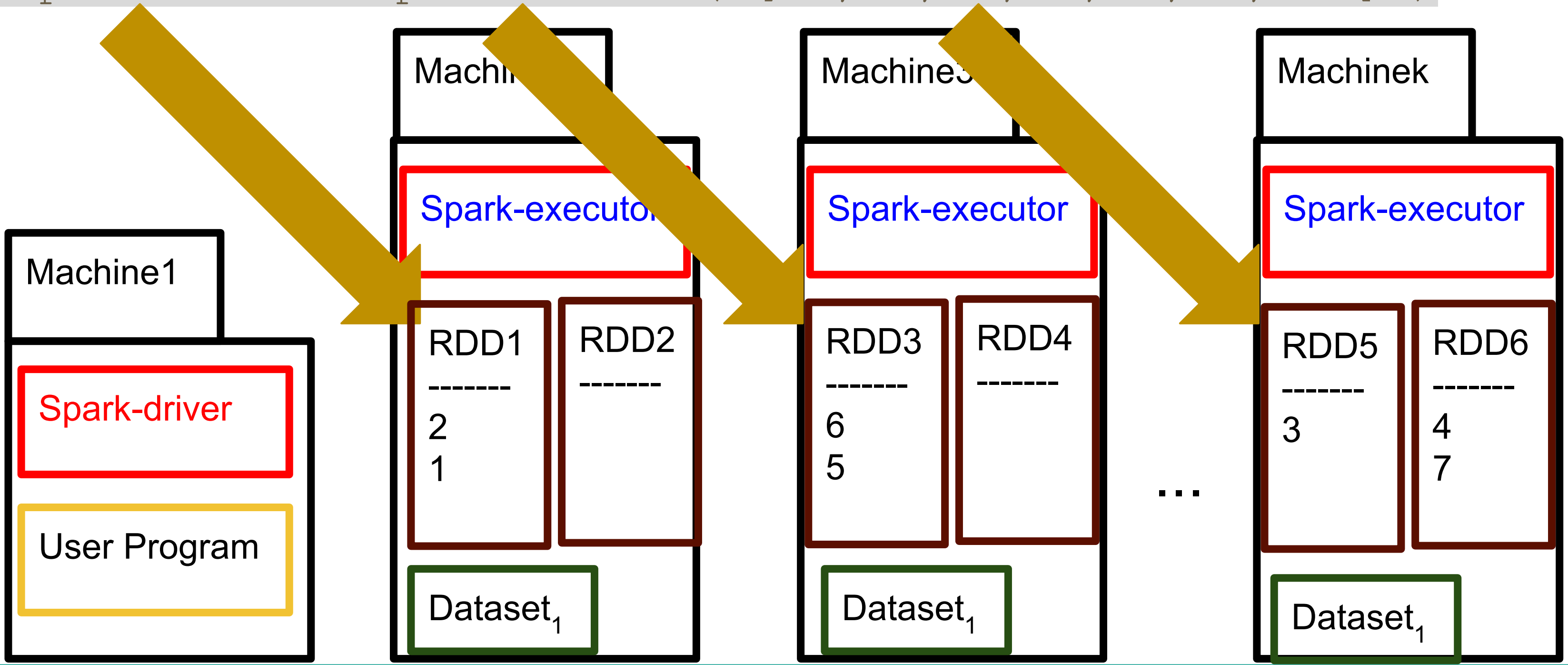
```
inputRDD  = sc.parallelize( [ 1, 2, 3, 4, 5, 6, 7 ] )
```



Partitions

- For example, given an inputRDD obtained from parallelizing a list:
  Or with this other distribution, again with 6 partitions, but where some partitions are empty.

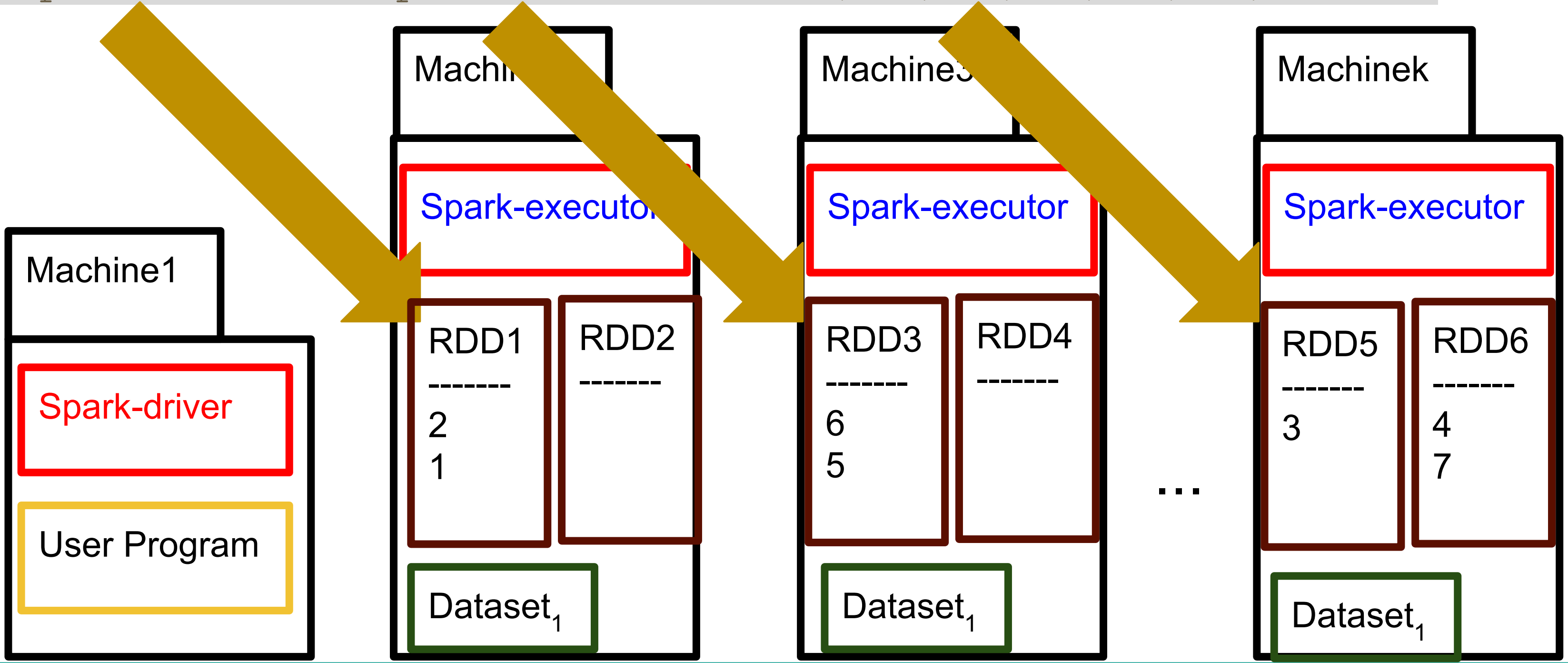
```
inputRDD  = sc.parallelize( [ 1, 2, 3, 4, 5, 6, 7 ] )
```



Partitions

- As we can see, a Spark executor process can host multiple partitions, but a partition cannot span across different executor processes.

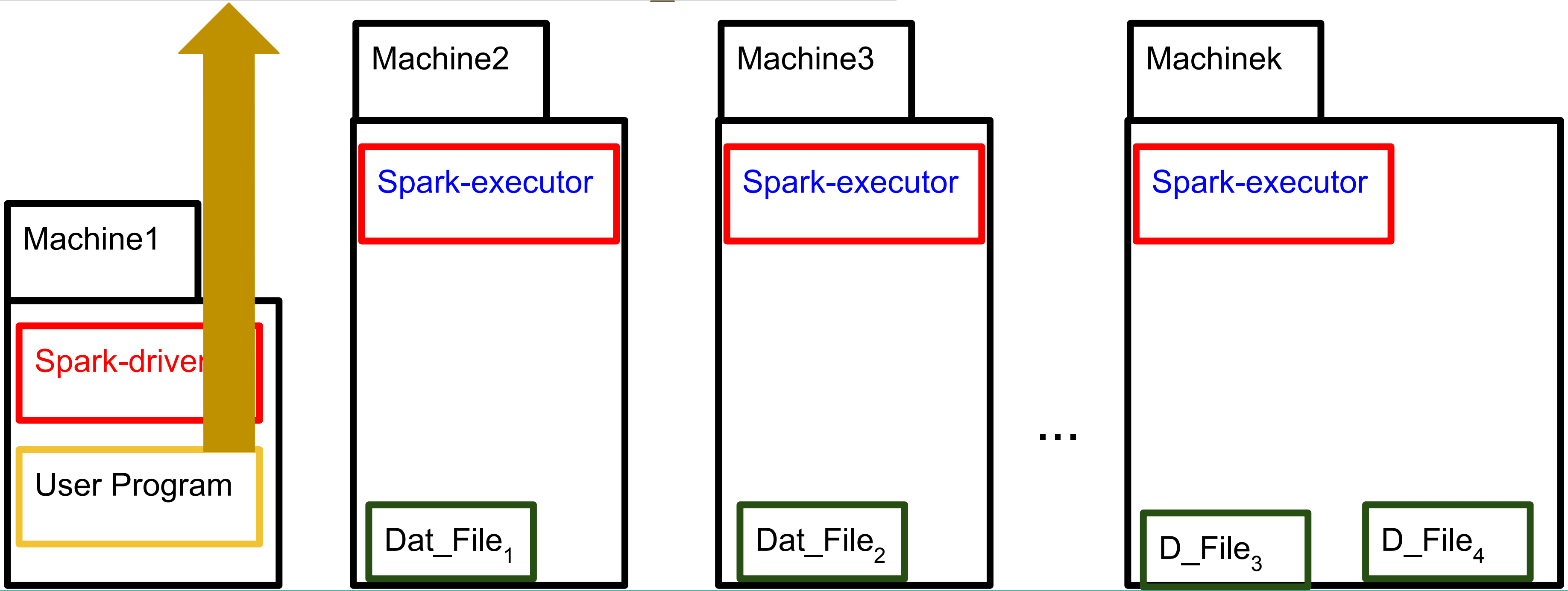
```
inputRDD  = sc.parallelize( [ 1, 2, 3, 4, 5, 6, 7 ] )
```



Partitions

➢ When getting an RDD from a dataset, one partition per file block is created (e.g., one partition per block of 64MB).

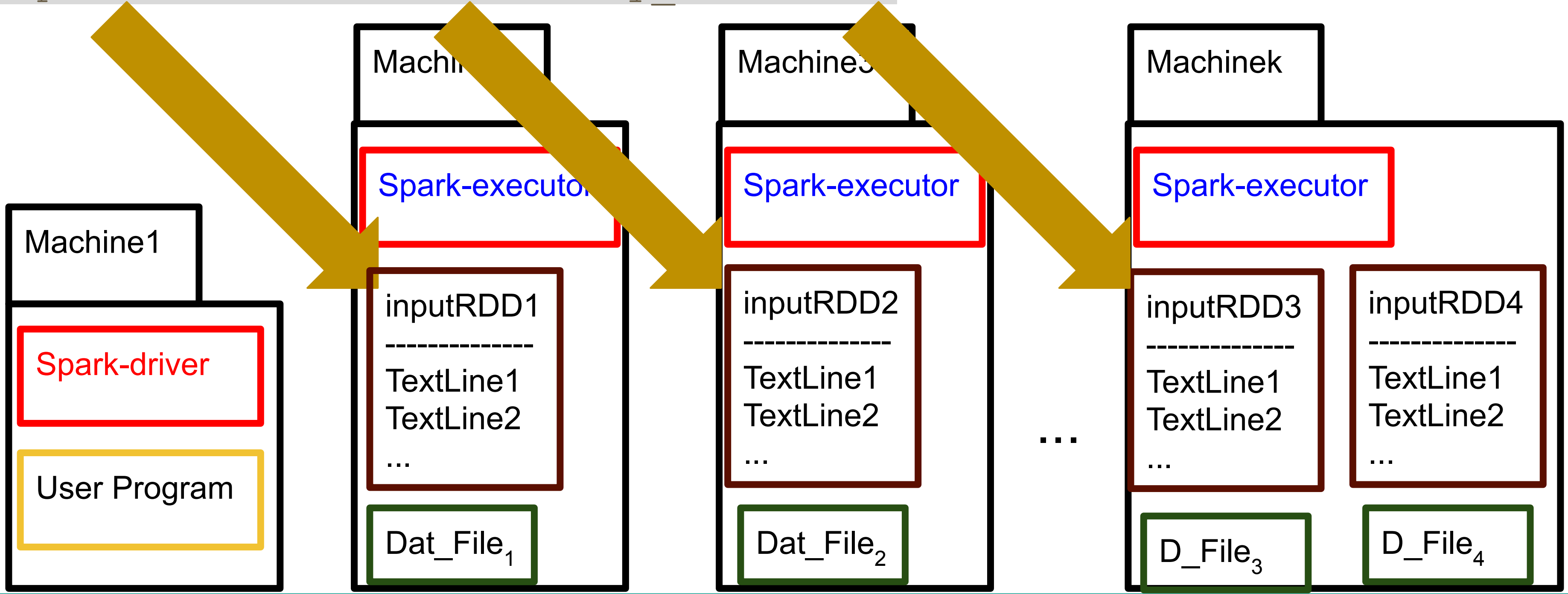
```
inputRDD  = sc.textFile( my_dataset )
```



Partitions

➢ When getting an RDD from a dataset, one partition per file block is created (e.g., one partition per block of 64MB).

```
inputRDD  = sc.textFile( my_dataset )
```



Outline

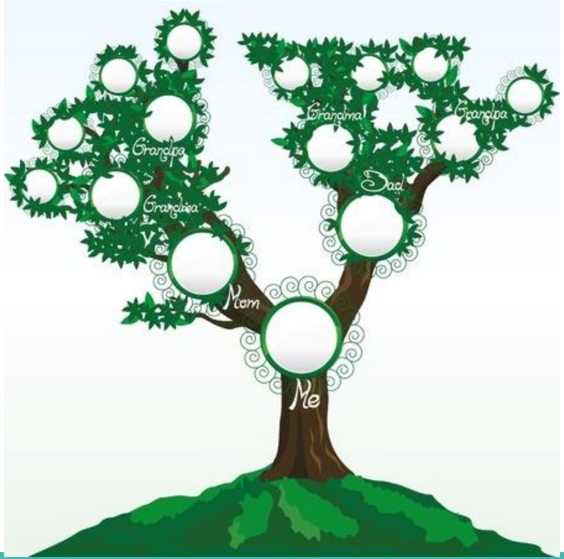
1. Setting the Context.
2. RDD Private Side: Partitions and Lineage.
   a. Internal Representation.
   b. Partitions.
   c. Lineage: Narrow and Wide Transformations.
   d. Lineage: Lazy evaluation.
   e. Lineage: Fault tolerant.
3. Spark Application: Jobs, Stages and Tasks.

Lineage: Narrow and Wide Transformations

Let’s go with the ADT private side:

3. **How** is the data internally represented?
   Specify the concrete data structures used to layout the data.

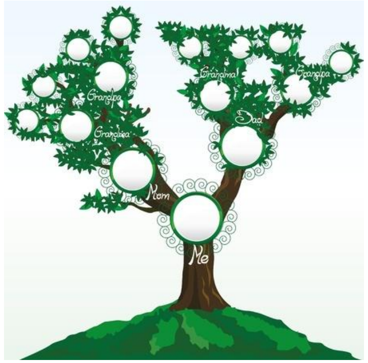
- An RDD is internally represented via…
  1. A set of **partitions**
  2. Enriched with **lineage** metadata for their reconstruction.



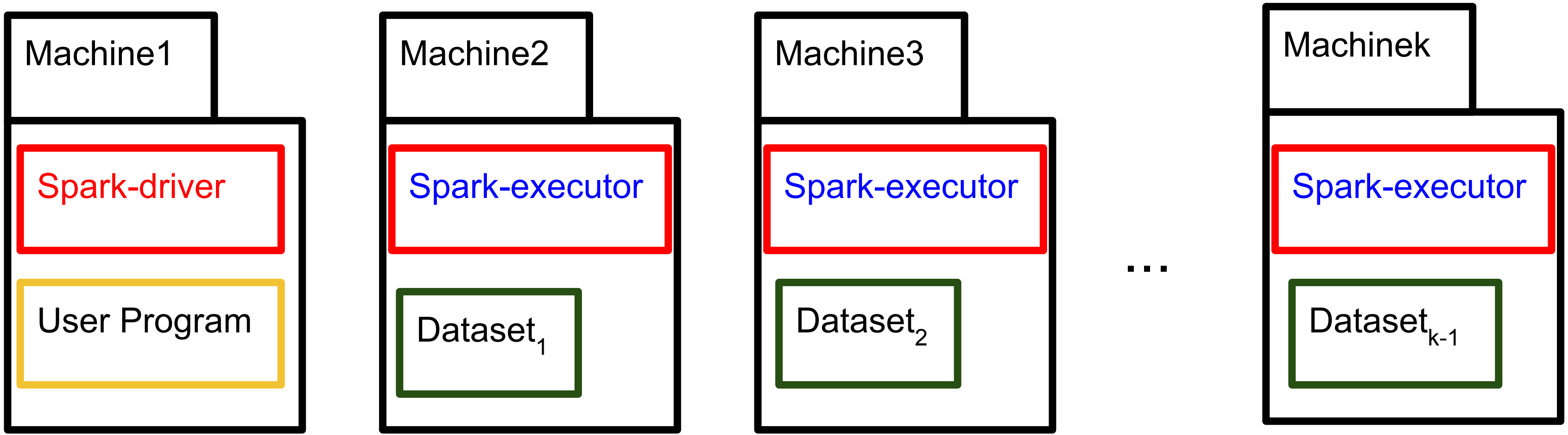
Lineage: Narrow and Wide Transformations

The motivation for having lineage metadata is more subtle.

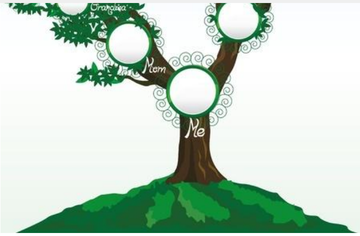
➢ However it is crucial for the lazy evaluation-based, fault tolerant computation model of Spark.



RDDs



Lineage: Narrow and Wide Transformations



The motivation for having lineage metadata is more subtle.

- Each time a **creation**, **transformation** or an **action** operation takes place, a dependency is created between the parent (original RDD/data source) and its child (novel RDD or result).

➢ For example, the following parallelize **creation** operation creates a dependency between inputRDD and the Spark driver.

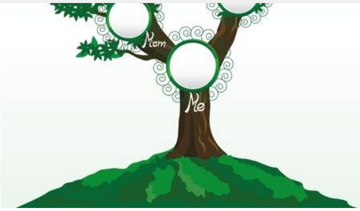
➢ For example, the following map **transformation** operation creates a dependency between inputRDD and newRDD.

```
inputRDD  = sc.parallelize( [ 1, 2, 3, 4, 5 ] )
newRDD = inputRDD.map( lambda elem : elem + 1 )
```

inputRDD → [1, 2, 3, 4, 5] or inputRDD → [5, 1, 3, 2, 4]

newRDD → [2, 3, 4, 5, 6] or newRDD → [6, 2, 4, 3, 5]

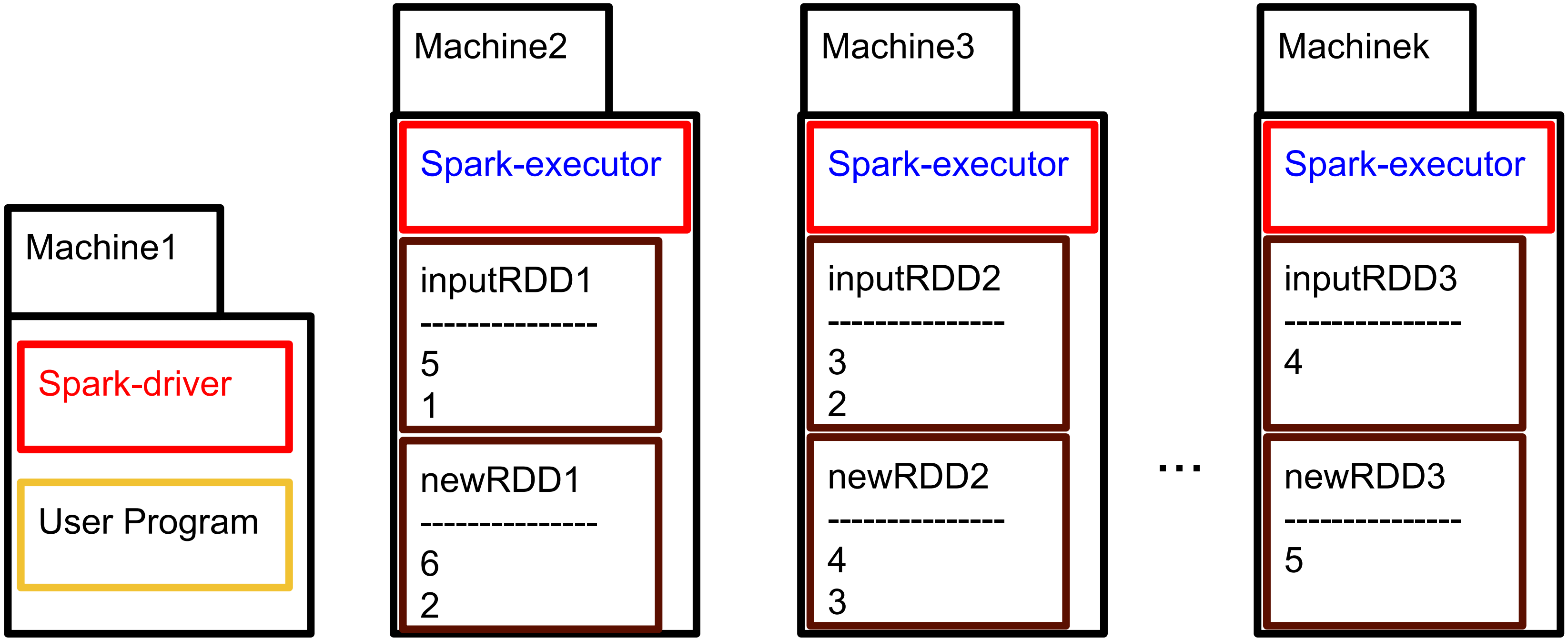
Lineage: Narrow and Wide Transformations



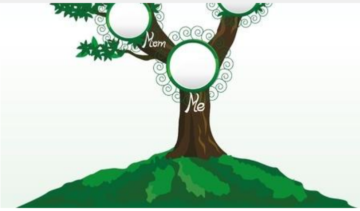
The motivation for having lineage metadata is more subtle.

- As RDDs are partitioned, dependencies are indeed among partitions:

```
inputRDD  = sc.parallelize( [ 1, 2, 3, 4, 5 ] )
newRDD = inputRDD.map( lambda elem : elem + 1 )
```



Lineage: Narrow and Wide Transformations

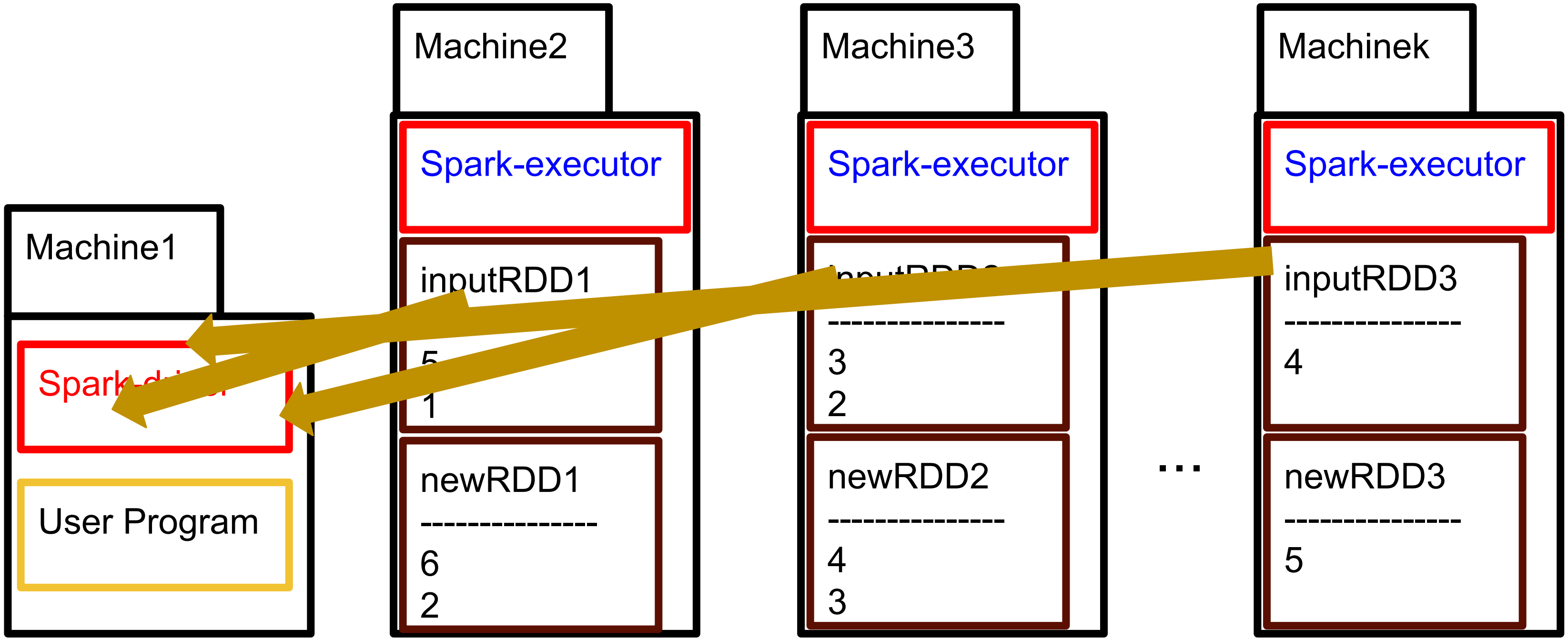


The motivation for having lineage metadata is more subtle.

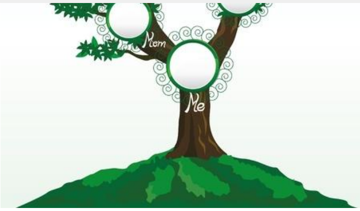
- As RDDs are partitioned, dependencies are indeed among partitions:

```
inputRDD  = sc.parallelize( [ 1, 2, 3, 4, 5 ] )
newRDD = inputRDD.map( lambda elem : elem + 1 )
```

There is a dependency between each inputRDD partition and the driver.



Lineage: Narrow and Wide Transformations

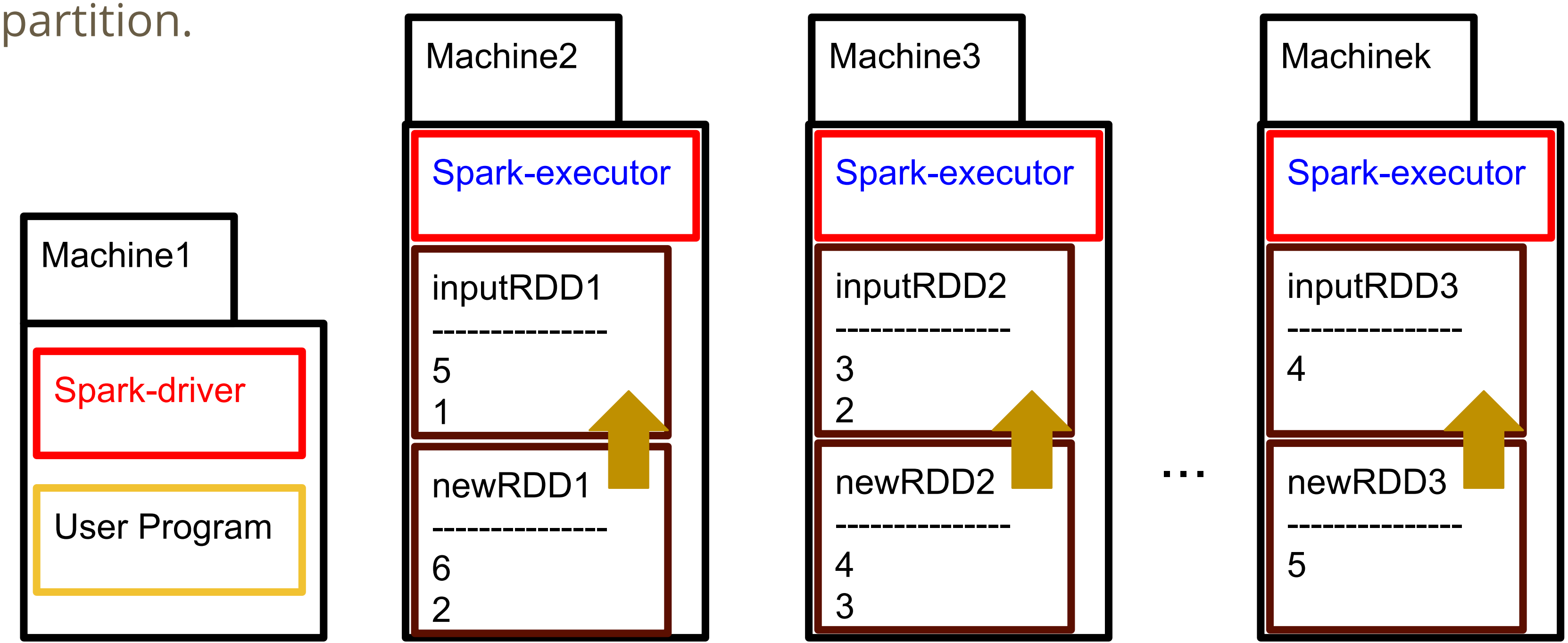


The motivation for having lineage metadata is more subtle.

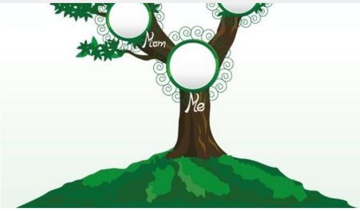
- As RDDs are partitioned, dependencies are indeed among partitions:

```
inputRDD  = sc.parallelize( [ 1, 2, 3, 4, 5 ] )
newRDD = inputRDD.map( lambda elem : elem + 1 )
```

There is a dependency between each newRDD partition and its parent inputRDD partition.



Lineage: Narrow and Wide Transformations

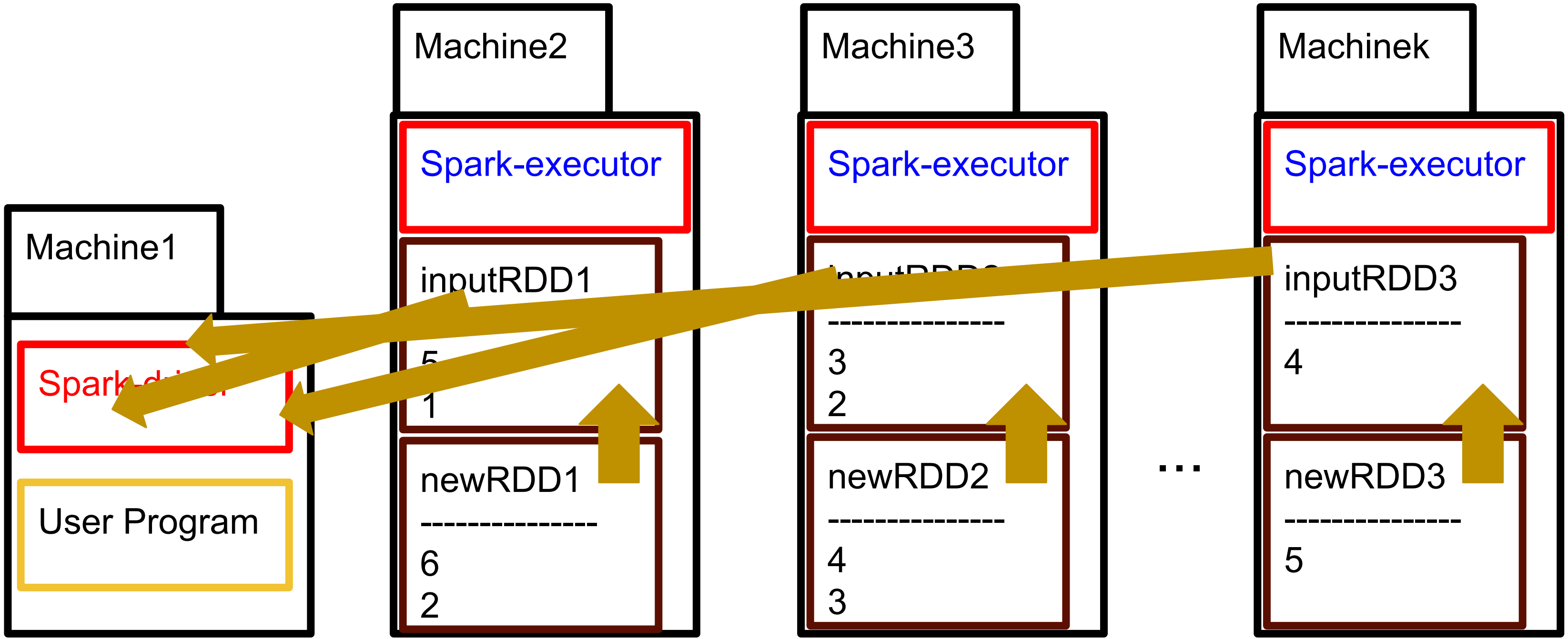


The motivation for having lineage metadata is more subtle.

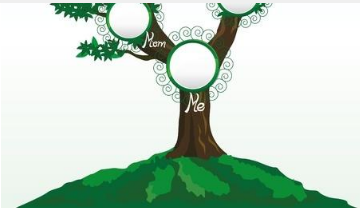
- As RDDs are partitioned, dependencies are indeed among partitions:

```
inputRDD  = sc.parallelize( [ 1, 2, 3, 4, 5 ] )
newRDD = inputRDD.map( lambda elem : elem + 1 )
```

Each of these two examples represent a narrow dependency!



Lineage: Narrow and Wide Transformations

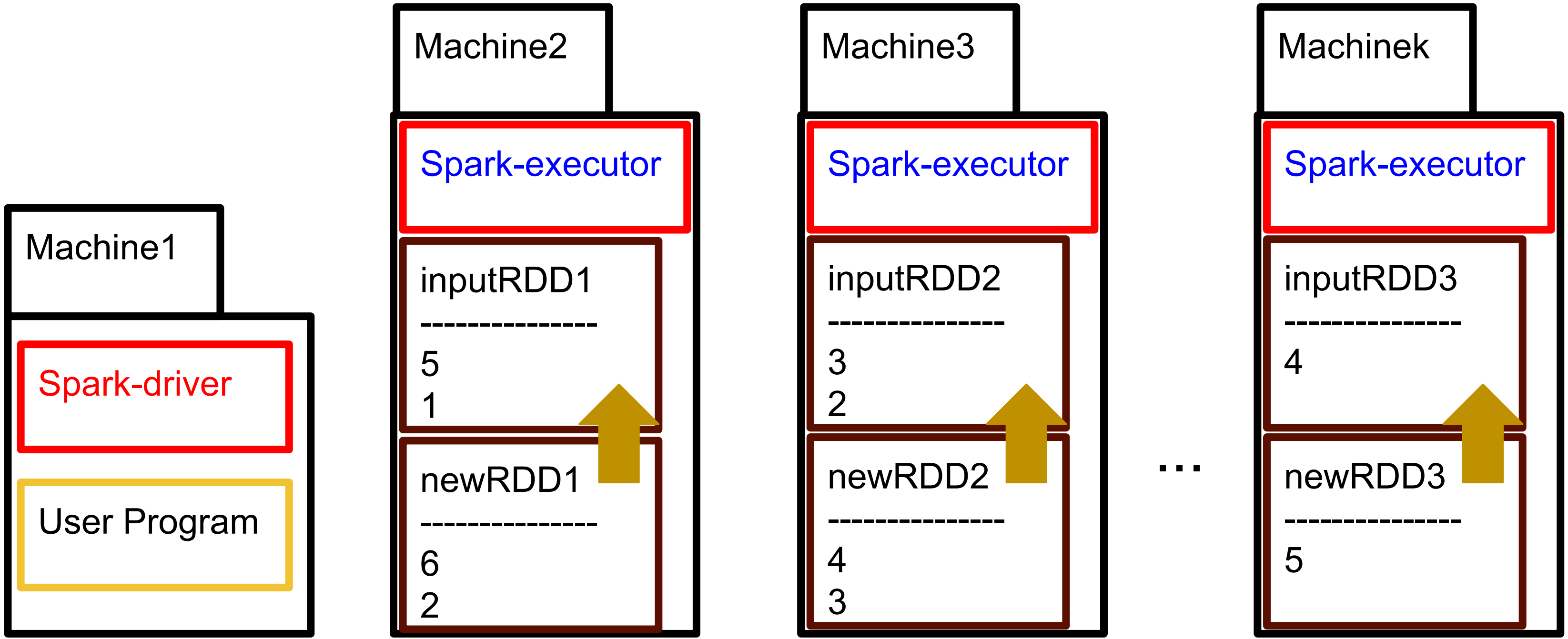


The motivation for having lineage metadata is more subtle.

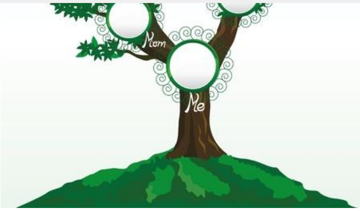
- As RDDs are partitioned, dependencies are indeed among partitions:

```
inputRDD  = sc.parallelize( [ 1, 2, 3, 4, 5 ] )
newRDD = inputRDD.map( lambda elem : elem + 1 )
```

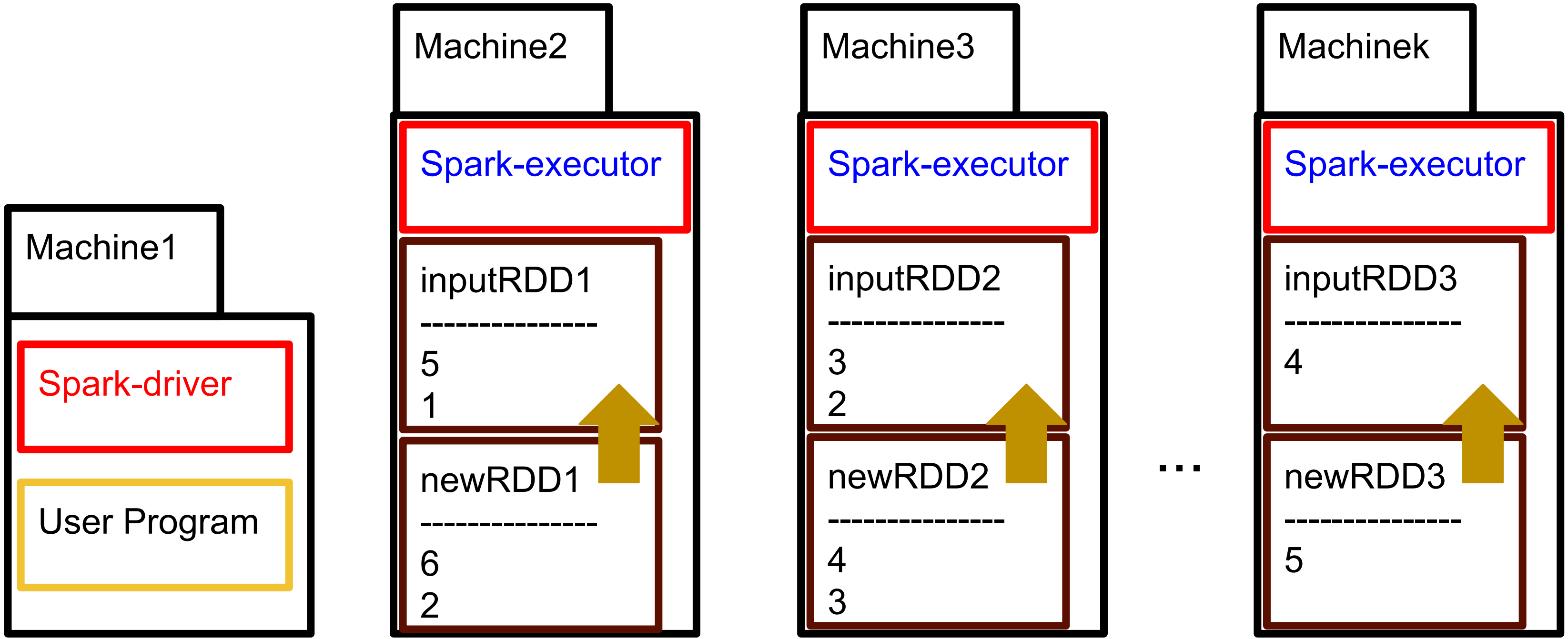
From now on let's just focus on the narrow dependency of the map, as the parallelize one makes the picture more messy with the diagonal arrows :)



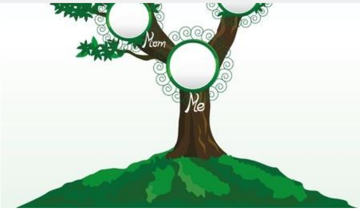
Lineage: Narrow and Wide Transformations



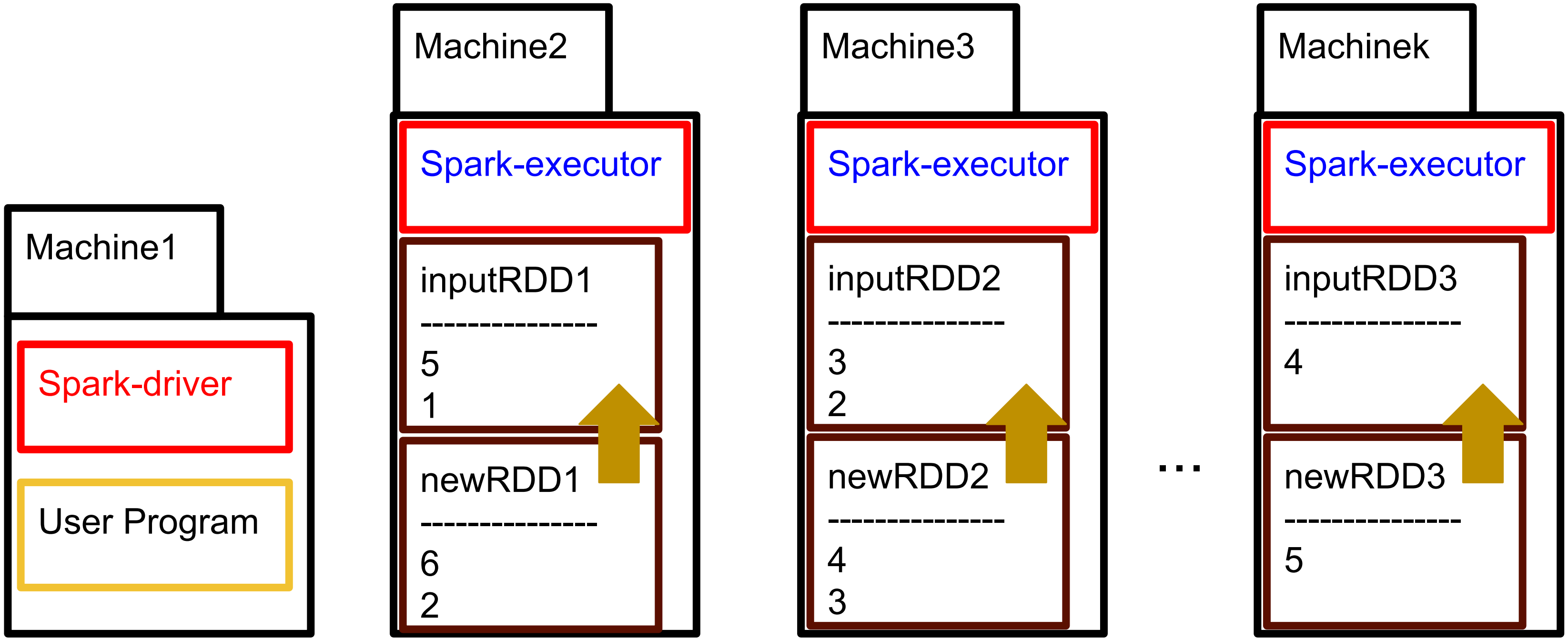
- Narrow dependency:
  1. Each partition in the child depends on 1 partition in the parent.



Lineage: Narrow and Wide Transformations



- Narrow dependency:
  1. Each partition in the child depends on 1 partition in the parent.
  2. The dependency can be determined at design time, irrespectively of the values hold by the parent partition.



Machine1

Spark-driver

User Program

Machine2

Spark-executor

inputRDD1

5
1

newRDD1

6
2

Machine3

Spark-executor

inputRDD2

3
2

newRDD2

4
3

…

Machinek

Spark-executor

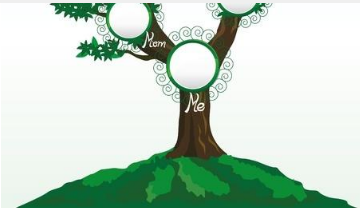
inputRDD3

4

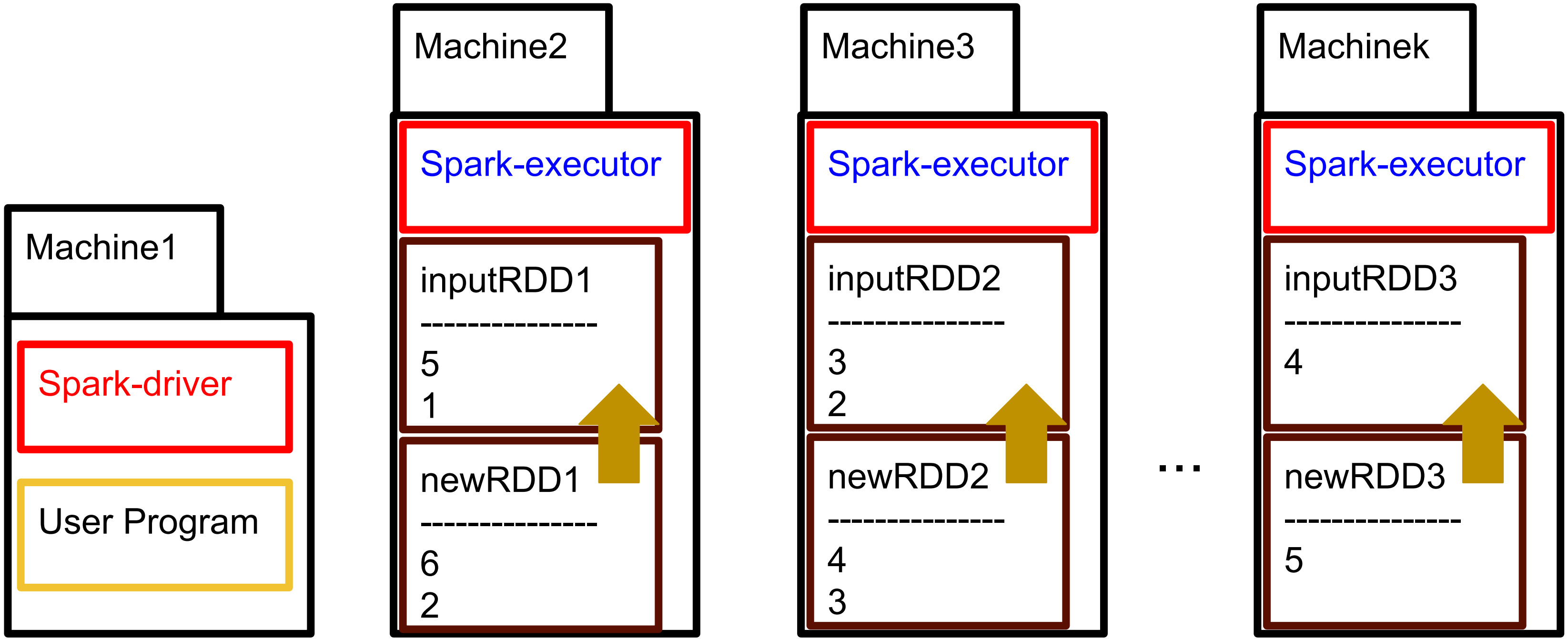
newRDD3

5

Lineage: Narrow and Wide Transformations



- Narrow dependency:
  1. Each partition in the child depends on 1 partition in the parent.
  2. The dependency can be determined at design time, irrespectively of the values hold by the parent partition.
  3. The transformation in one partition can be executed without any information about the other partitions.



Machine1

Spark-driver

User Program

Machine2

Spark-executor

inputRDD1

5
1

newRDD1

6
2

Machine3

Spark-executor

inputRDD2

3
2

newRDD2

4
3

…

Machinek

Spark-executor

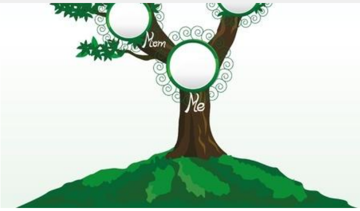
inputRDD3

4

newRDD3

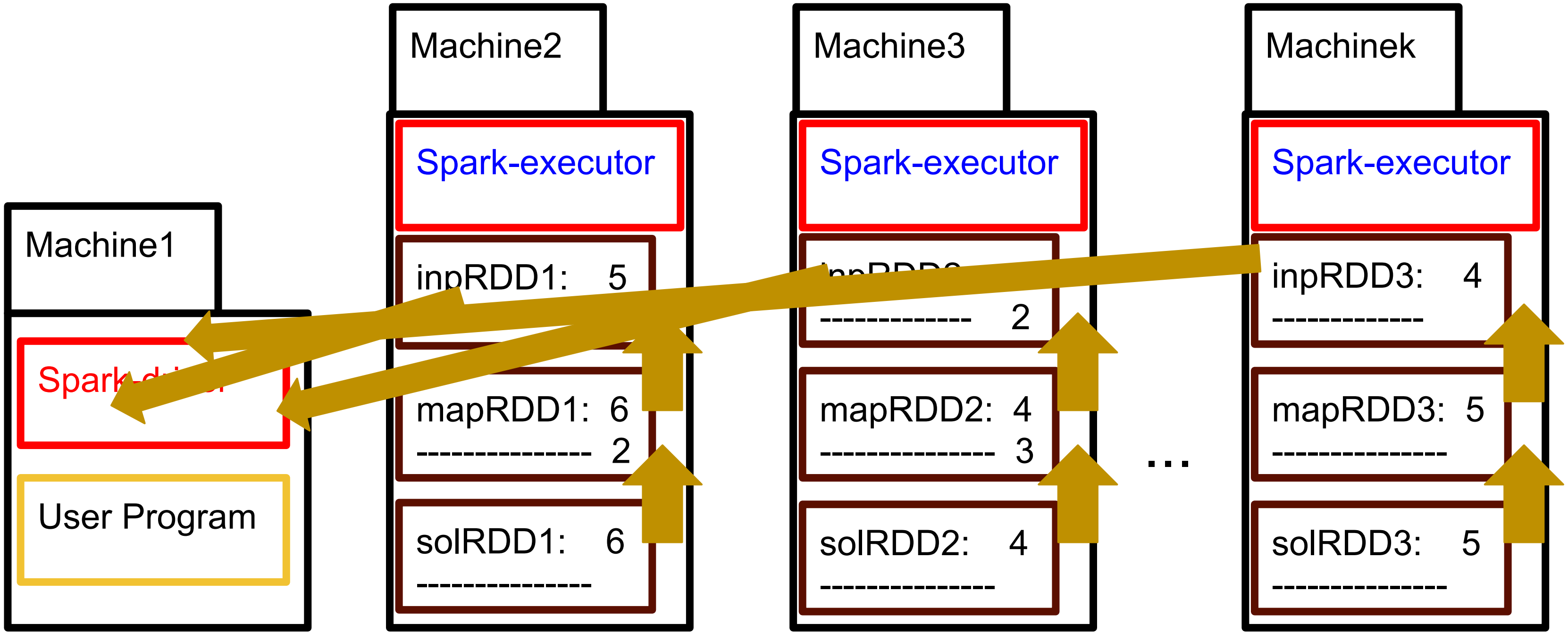
5

Lineage: Narrow and Wide Transformations

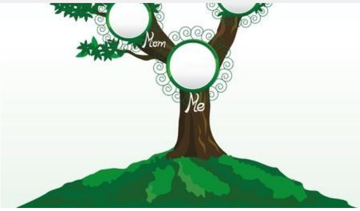


- Narrow dependency:
  3. The transformation in one partition can be executed without any information about the other partitions:
     ➢ This can indeed include multiple chained operations!

```
inputRDD  = sc.parallelize( [ 1, 2, 3, 4, 5 ] )
mapRDD = inputRDD.map( lambda elem : elem + 1 )
solRDD = mapRDD.filter( lambda elem : elem > 3 )
```



Lineage: Narrow and Wide Transformations



The motivation for having lineage metadata is more subtle.

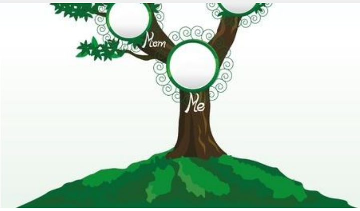
- Each time a **creation**, **transformation** or an **action** operation takes place, a dependency is created between the parent (original RDD) and its child (novel RDD or result).

➢ For example, the following reduceByKey transformation creates a dependency between inputRDD and newRDD:

```
inputRDD  = sc.parallelize([(1,1), (2,4), (1,3), (2,5), (1,6)])
newRDD = inputRDD.reduceByKey( lambda x,y : x + y )
```

inputRDD →[(1,1), (2,4), (1,3), (2,5), (1,6)] or inputRDD →[(2,4), (1,1), (1,3), (1,6), (2,5)]
newRDD → [(1,10), (2,9)] or newRDD → [(2,9), (1,10)]

Lineage: Narrow and Wide Transformations

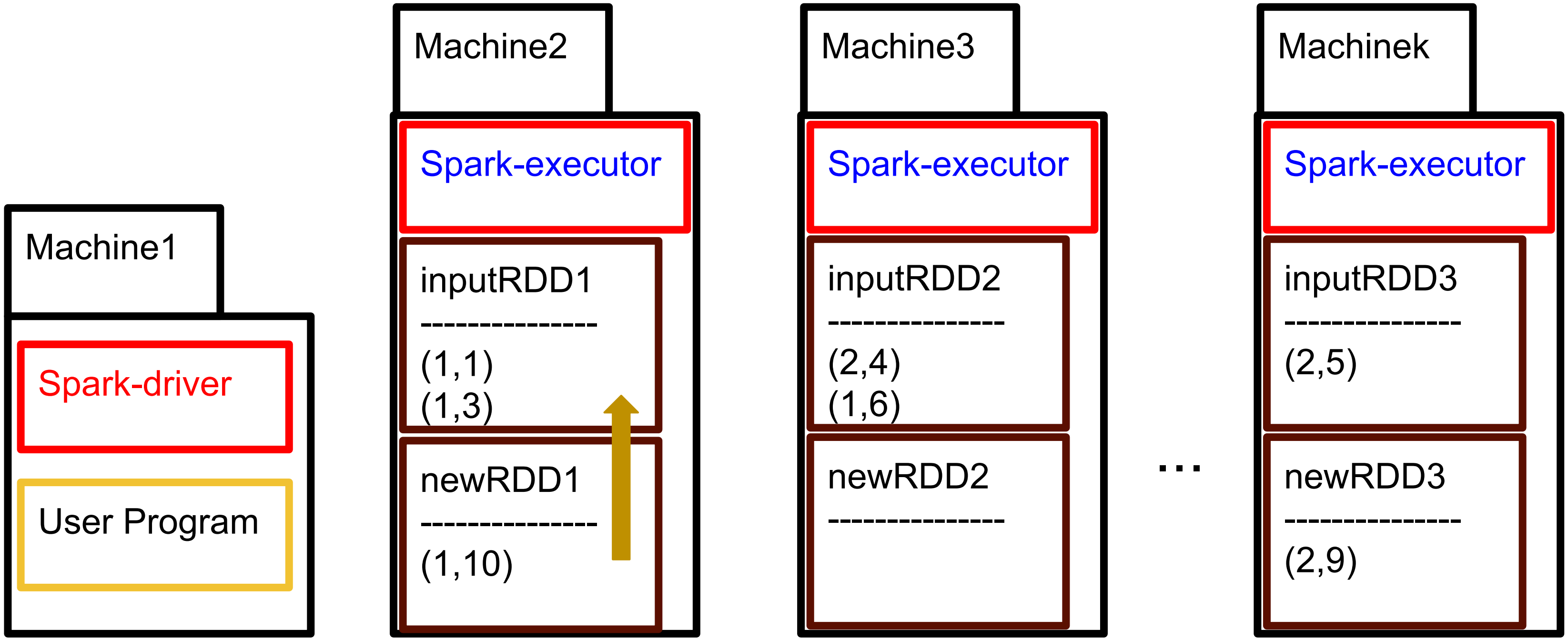


The motivation for having lineage metadata is more subtle.

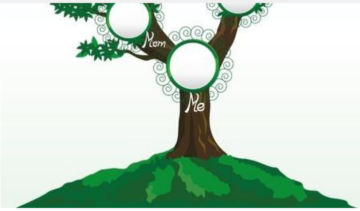
- As RDDs are partitioned, dependencies are indeed among partitions:

```
inputRDD  = sc.parallelize([(1,1), (2,4), (1,3), (2,5), (1,6)])
newRDD = inputRDD.reduceByKey( lambda x, y : x + y )
```

But here the dependency for newRDD1 partition is on its parent newRDD1 partition



Lineage: Narrow and Wide Transformations

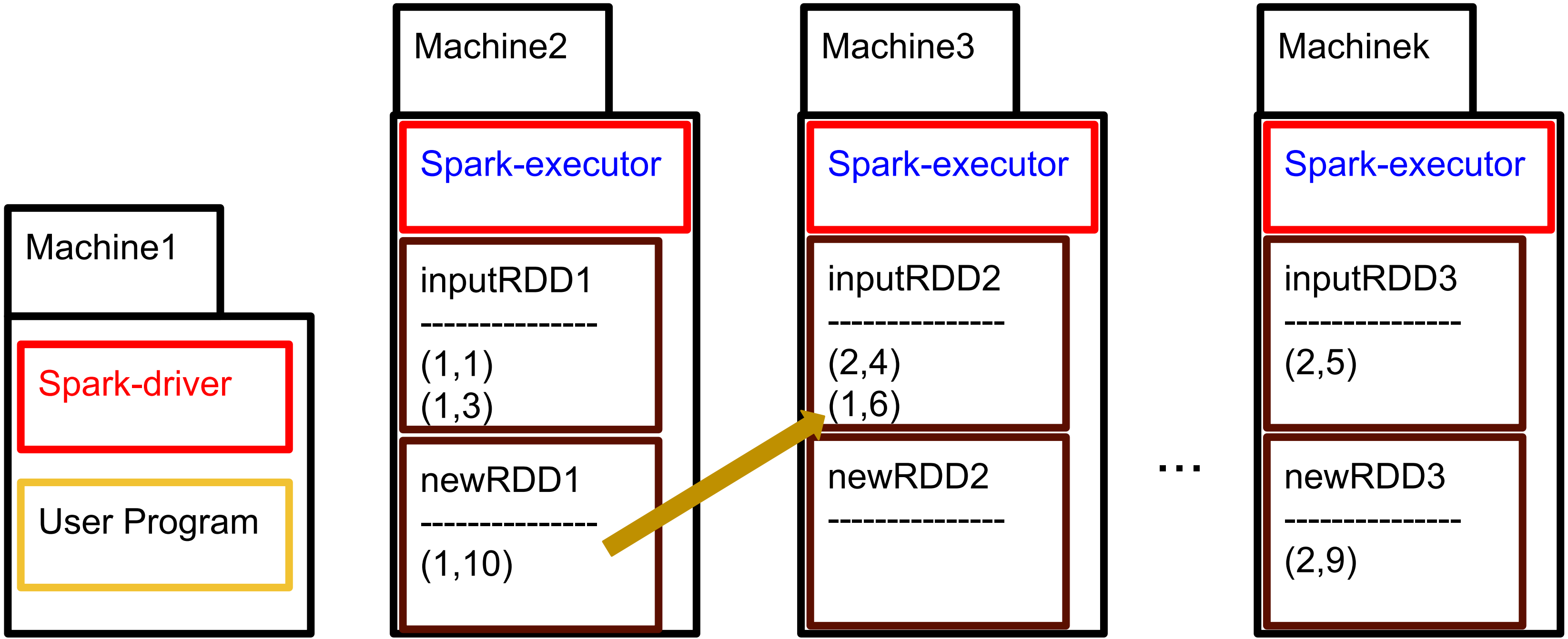


The motivation for having lineage metadata is more subtle.

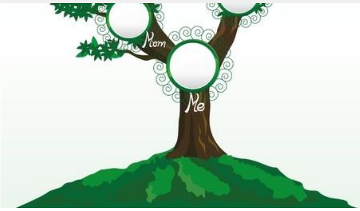
- As RDDs are partitioned, dependencies are indeed among partitions:

```
inputRDD  = sc.parallelize([(1,1), (2,4), (1,3), (2,5), (1,6)])
newRDD = inputRDD.reduceByKey( lambda x, y : x + y )
```

...and in its parent inputRDD2 partition...



Lineage: Narrow and Wide Transformations

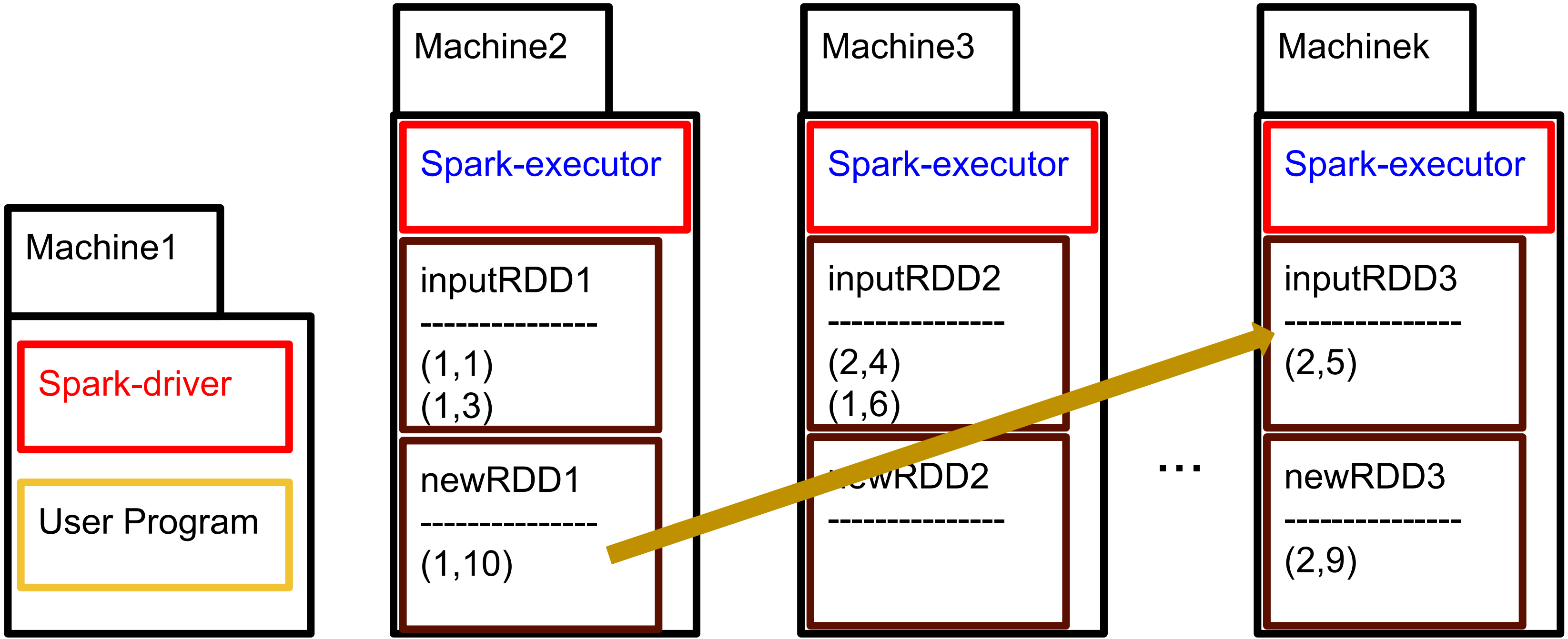


The motivation for having lineage metadata is more subtle.

- As RDDs are partitioned, dependencies are indeed among partitions:

```
inputRDD  = sc.parallelize([(1,1), (2,4), (1,3), (2,5), (1,6)])
newRDD = inputRDD.reduceByKey( lambda x, y : x + y )
```

...and on its parent inputRDD3 partition as well.



Machine1

Spark-driver

User Program

Machine2

Spark-executor

inputRDD1

(1,1)
(1,3)

newRDD1

(1,10)

Machine3

Spark-executor

inputRDD2

(2,4)
(1,6)

newRDD2

...

Machinek

Spark-executor

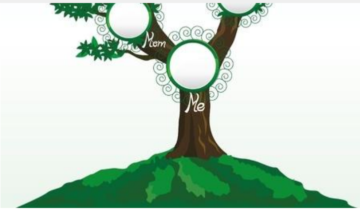
inputRDD3

(2,5)

newRDD3

(2,9)

Lineage: Narrow and Wide Transformations

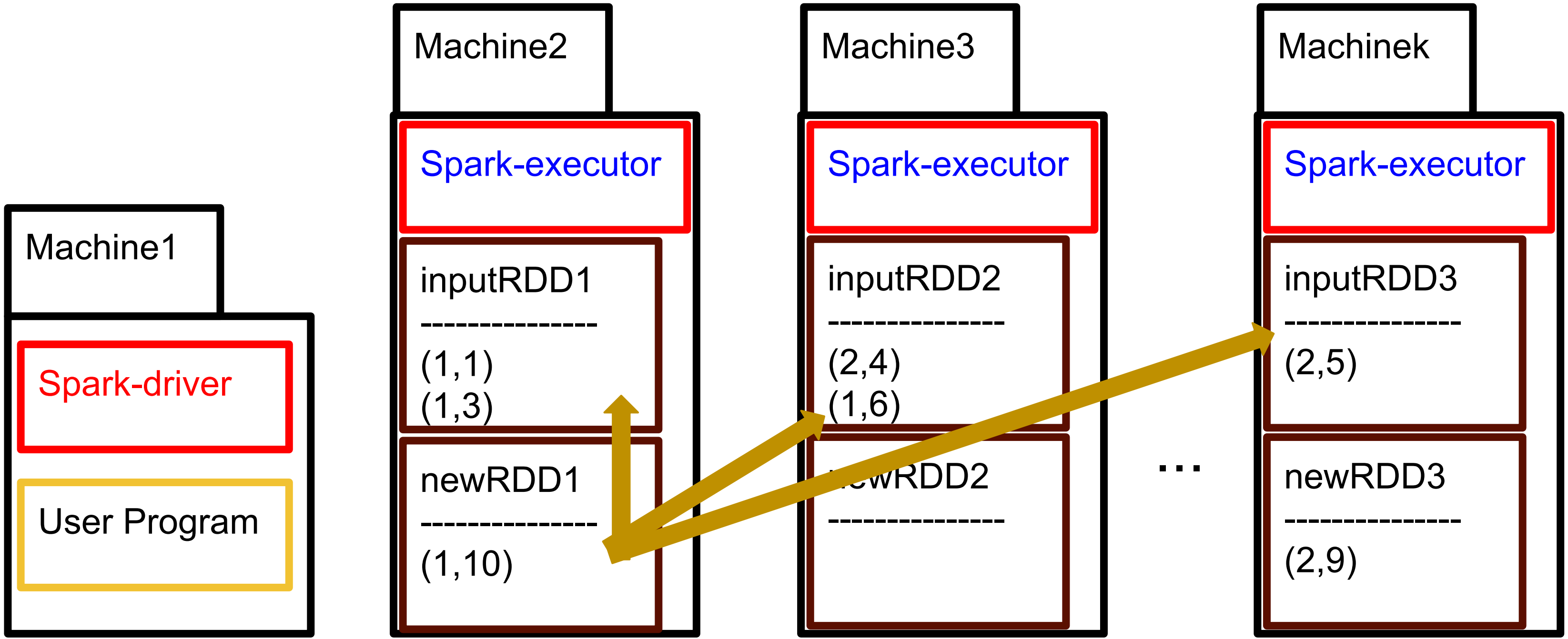


The motivation for having lineage metadata is more subtle.

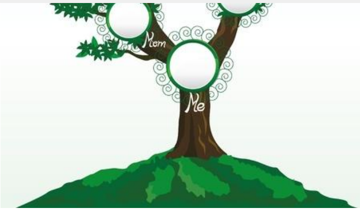
- As RDDs are partitioned, dependencies are indeed among partitions:

```
inputRDD  = sc.parallelize([(1,1), (2,4), (1,3), (2,5), (1,6)])
newRDD = inputRDD.reduceByKey( lambda x, y : x + y )
```

So, as we can see, this partition depends in many parent partitions.
In this case, even in all of them.



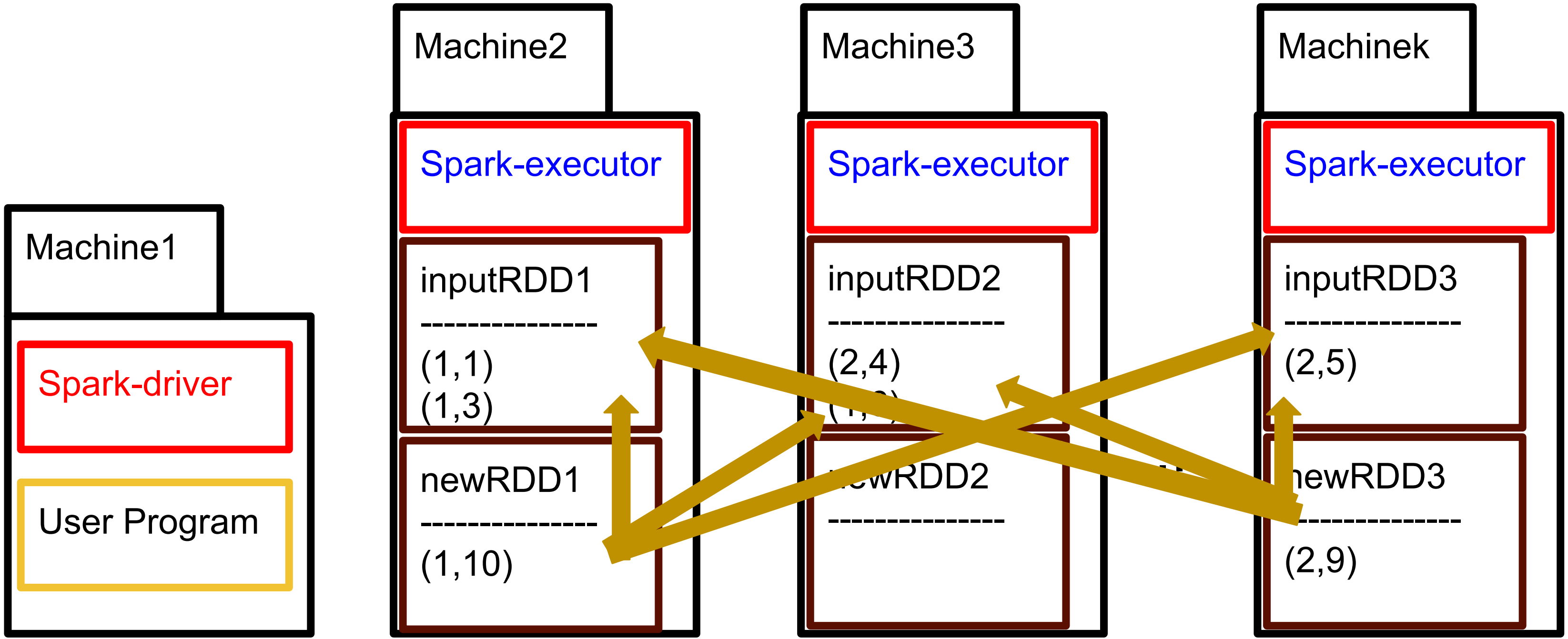
Lineage: Narrow and Wide Transformations



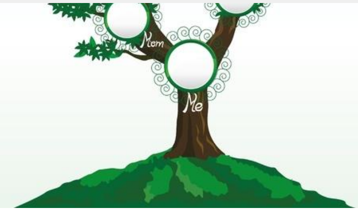
The motivation for having lineage metadata is more subtle.

- As RDDs are partitioned, dependencies are indeed among partitions:

```
inputRDD  = sc.parallelize([(1,1), (2,4), (1,3), (2,5), (1,6)])
newRDD = inputRDD.reduceByKey( lambda x, y : x + y )
```



Lineage: Narrow and Wide Transformations

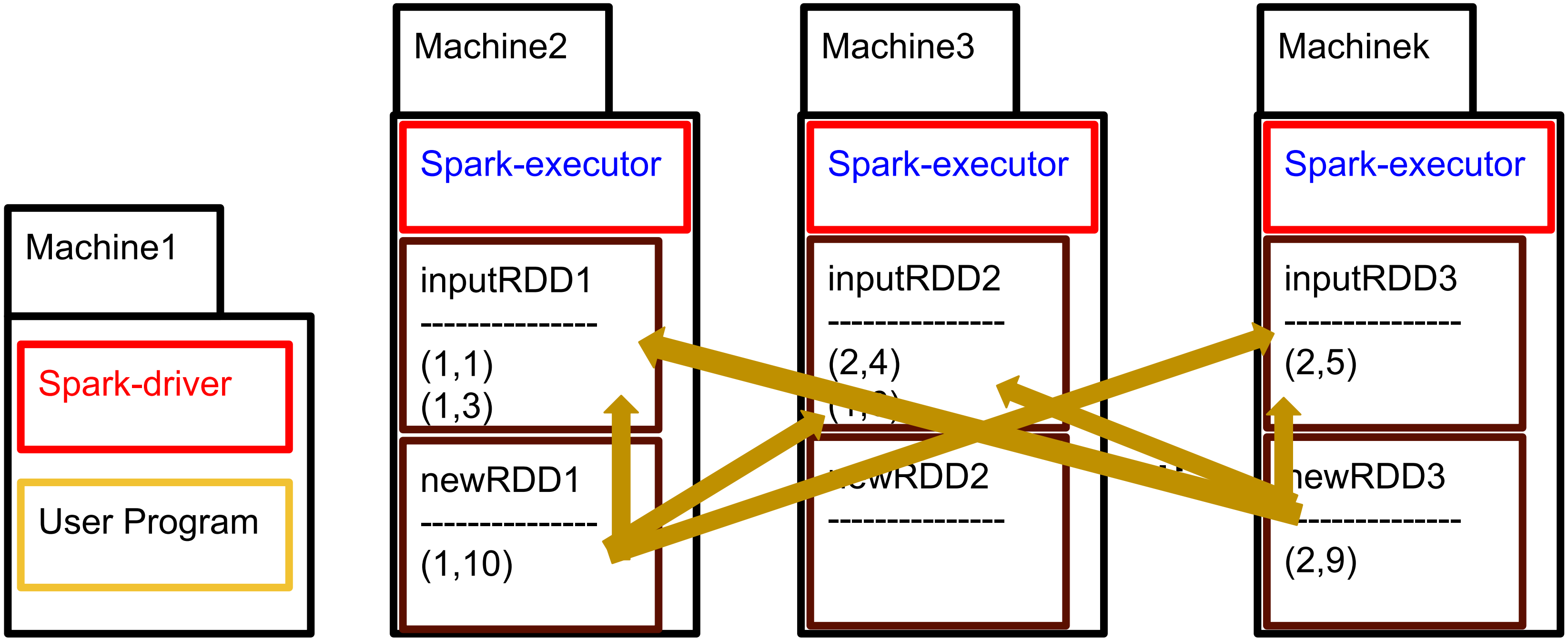


The motivation for having lineage metadata is more subtle.

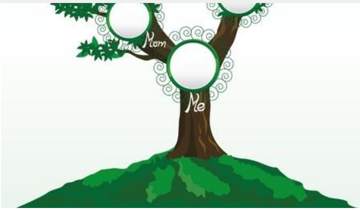
- As RDDs are partitioned, dependencies are indeed among partitions:

```
inputRDD  = sc.parallelize([(1,1), (2,4), (1,3), (2,5), (1,6)])
newRDD = inputRDD.reduceByKey( lambda x, y : x + y )
```

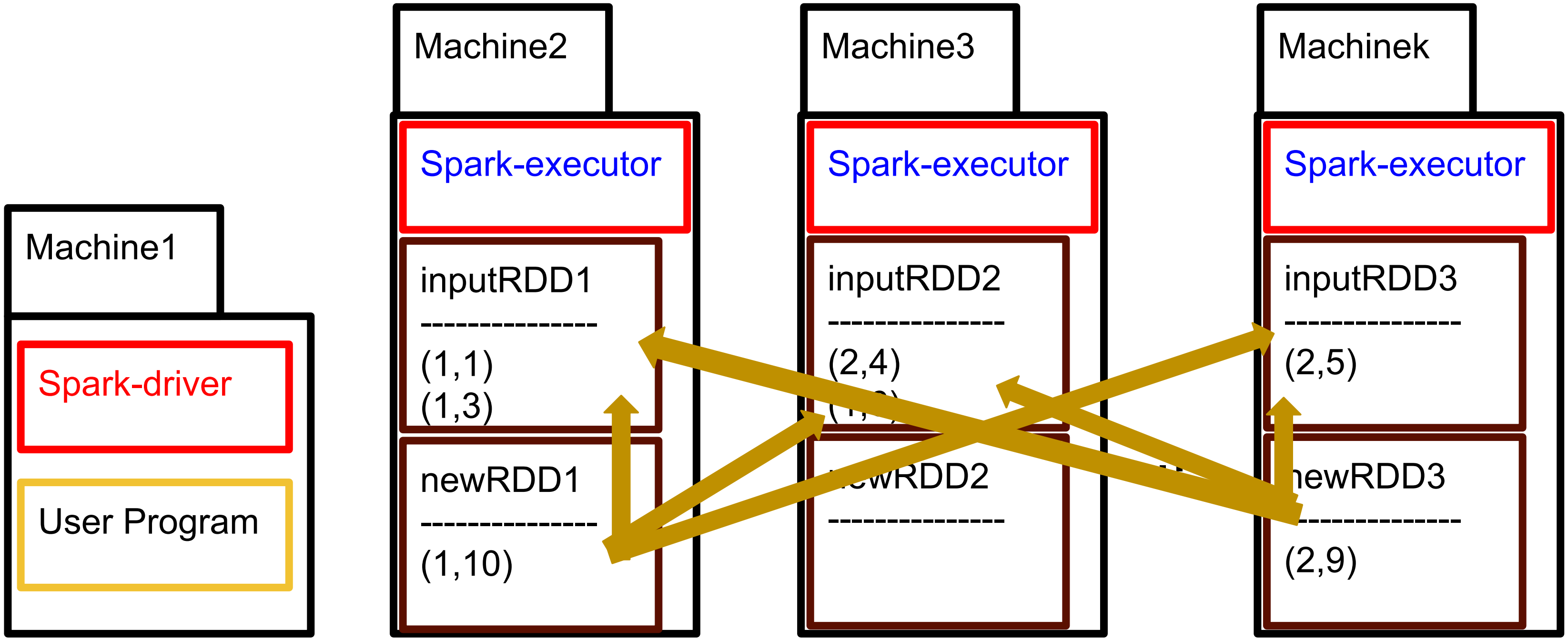
This is a wide dependency!



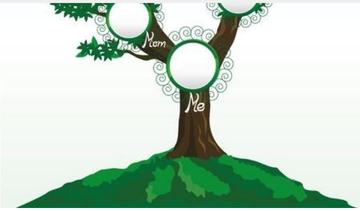
Lineage: Narrow and Wide Transformations



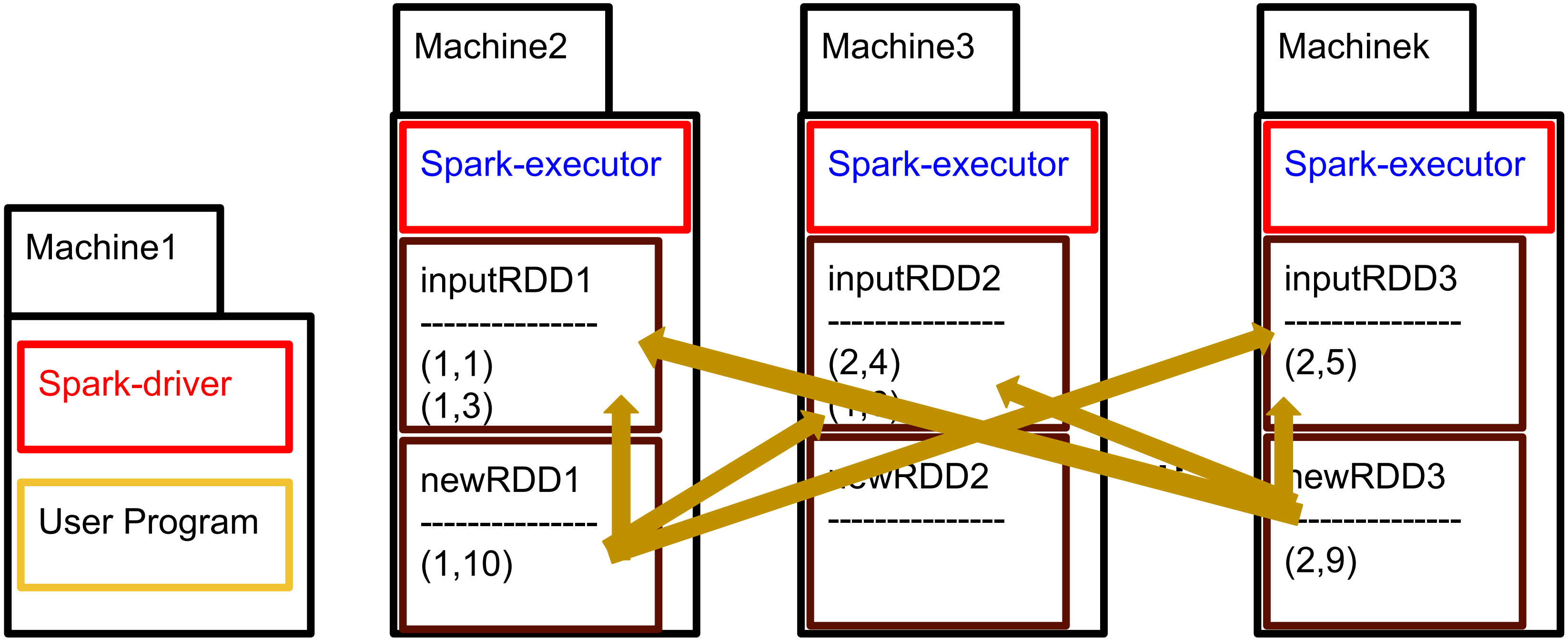
- Wide dependency:
  1. A child partition depends on an arbitrary set of parent partitions.



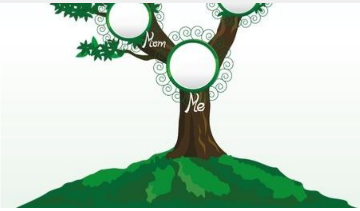
Lineage: Narrow and Wide Transformations



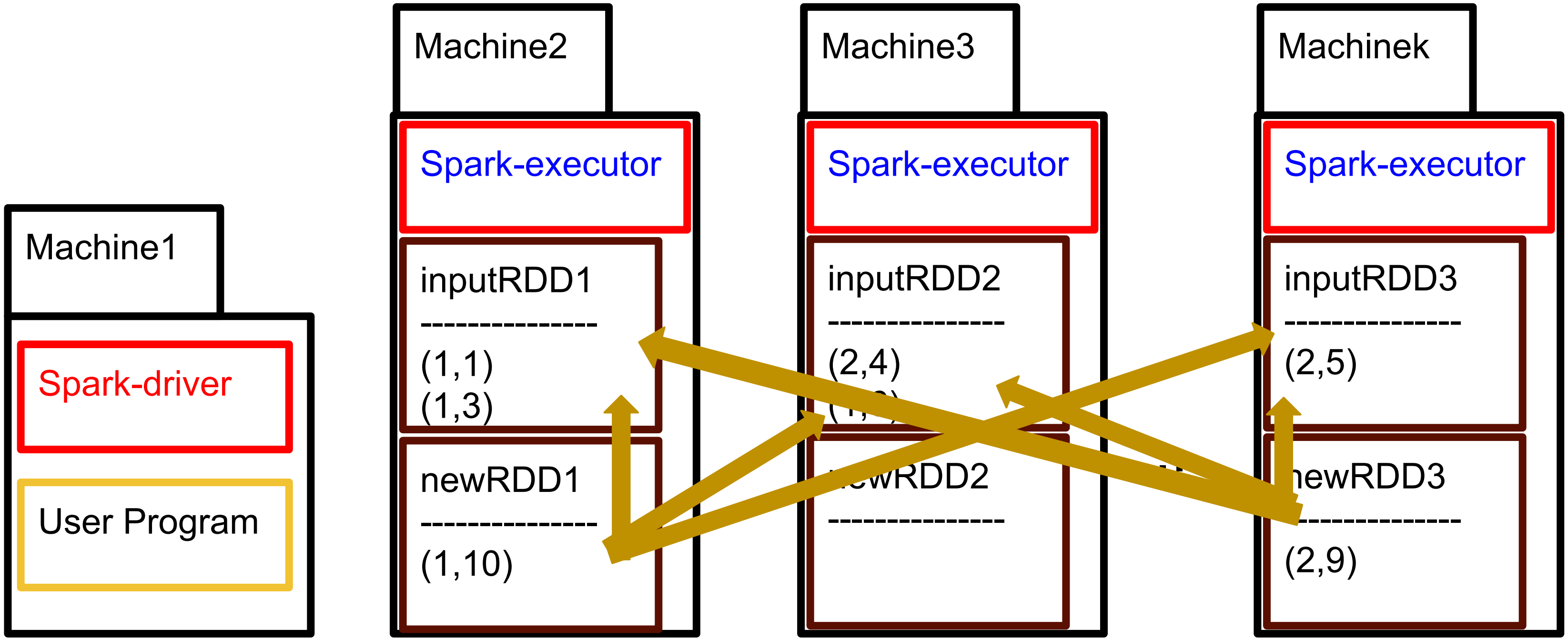
- Wide dependency:
  1. A child partition depends on an arbitrary set of parent partitions.
  2. These dependencies are determined by the concrete values of a partition, and thus cannot be known at design time (before data is evaluated).



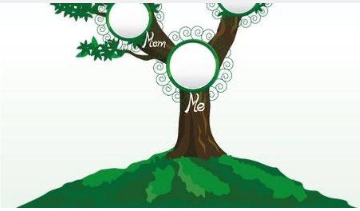
Lineage: Narrow and Wide Transformations



- Wide dependency:
  1. A child partition depends on an arbitrary set of parent partitions.
  2. These dependencies are determined by the concrete values of a partition, and thus cannot be known at design time (before data is evaluated).
  3. As data requires to be shuffled, a transformation in one partition cannot be executed without any information about the other partitions.

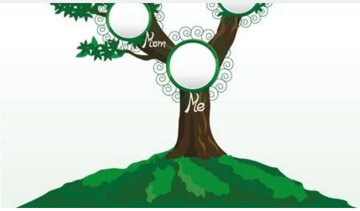


Lineage: Narrow and Wide Transformations



- Wide dependency:
  3. As data requires to be shuffled, a transformation in one partition cannot be executed without any information about the other partitions:
     - ➢ This breaks the possibility of a partition executing multiple chained transformations on its own!

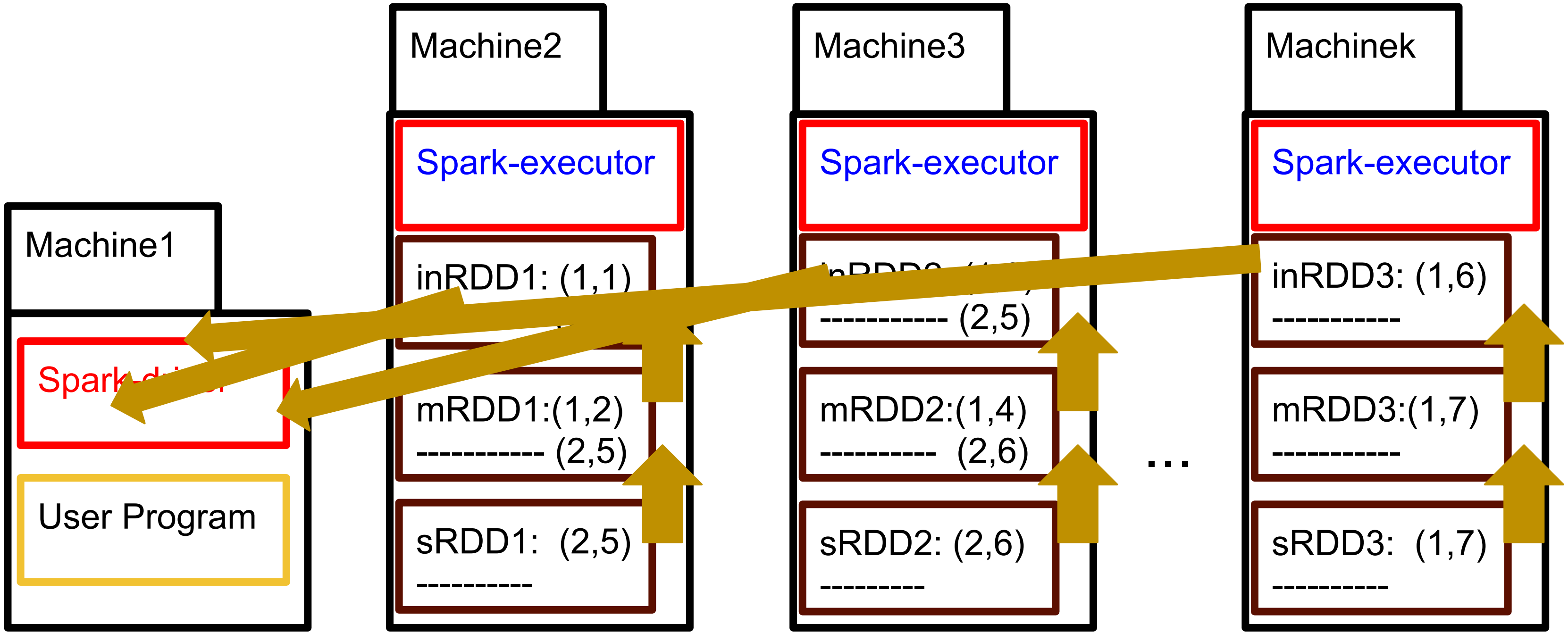
Lineage: Narrow and Wide Transformations



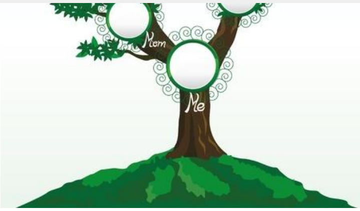
- Narrow vs Wide dependencies:

In this example, as parallelize, map and filter are narrow dependencies, they can be chained on each partition working on its own.

```
inputRDD  = sc.parallelize([(1,1), (2,4), (1,3), (2,5), (1,6)])
mapRDD = inputRDD.map( lambda elem : elem[1] + 1 )
solRDD = mapRDD.filter( lambda elem : elem[1] >= 5 )
```



Lineage: Narrow and Wide Transformations

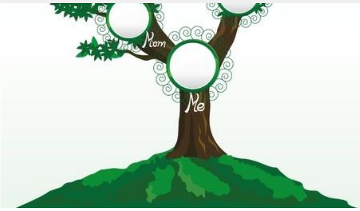


- Narrow vs Wide dependencies:

But, in this new example, as reduceByKey is a wide dependency, it breaks the set of operations to be chained by each partition on its own.

```
inputRDD  = sc.parallelize([(1,1), (2,4), (1,3), (2,5), (1,6)])
mapRDD = inputRDD.map( lambda elem : elem[1] + 1 )
redRDD = mapRDD.reduceByKey( lambda x, y: x + y )
solRDD = mapRDD.filter( lambda elem : elem[1] > 9 )
```

Lineage: Narrow and Wide Transformations

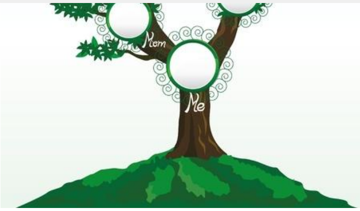


- Narrow vs Wide dependencies:

First, parallelize and map can be done on its own by each partition.

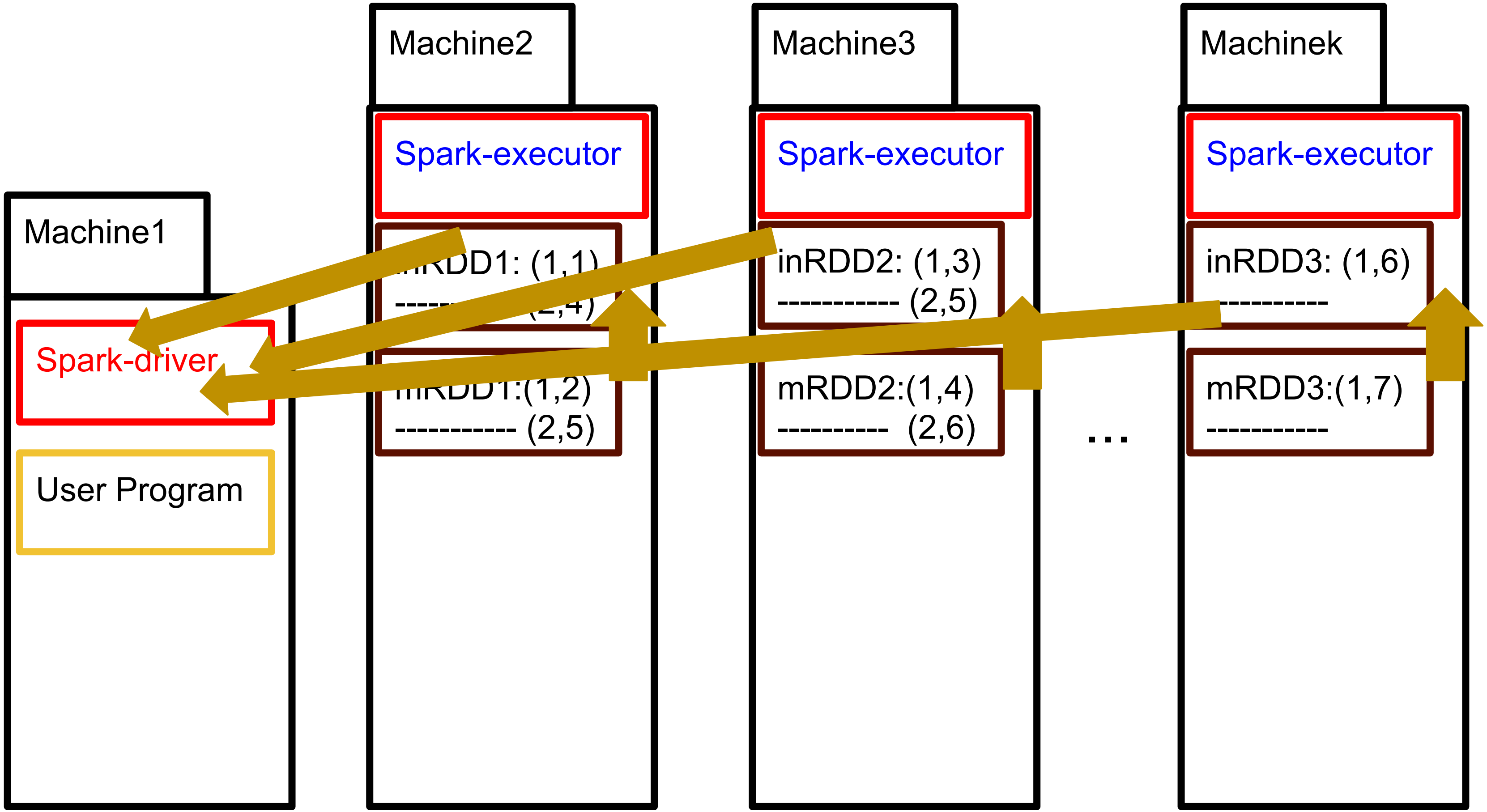
```
inputRDD  = sc.parallelize([(1,1), (2,4), (1,3), (2,5), (1,6)])
mapRDD = inputRDD.map( lambda elem : elem[1] + 1 )
redRDD = mapRDD.reduceByKey( lambda x, y: x + y )
solRDD = mapRDD.filter( lambda elem : elem[1] > 9 )
```

Lineage: Narrow and Wide Transformations

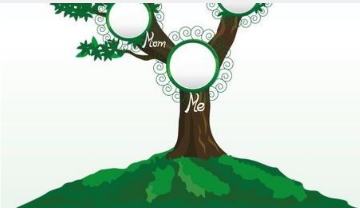


- Narrow vs Wide dependencies:

First, the transformation map can be done on its own by each partition.



Lineage: Narrow and Wide Transformations

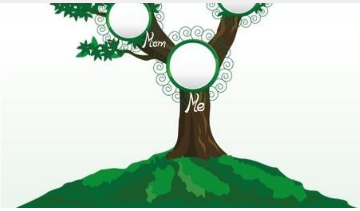


- Narrow vs Wide dependencies:

Second, the transformation reduceByKey shuffles the data among the partitions.

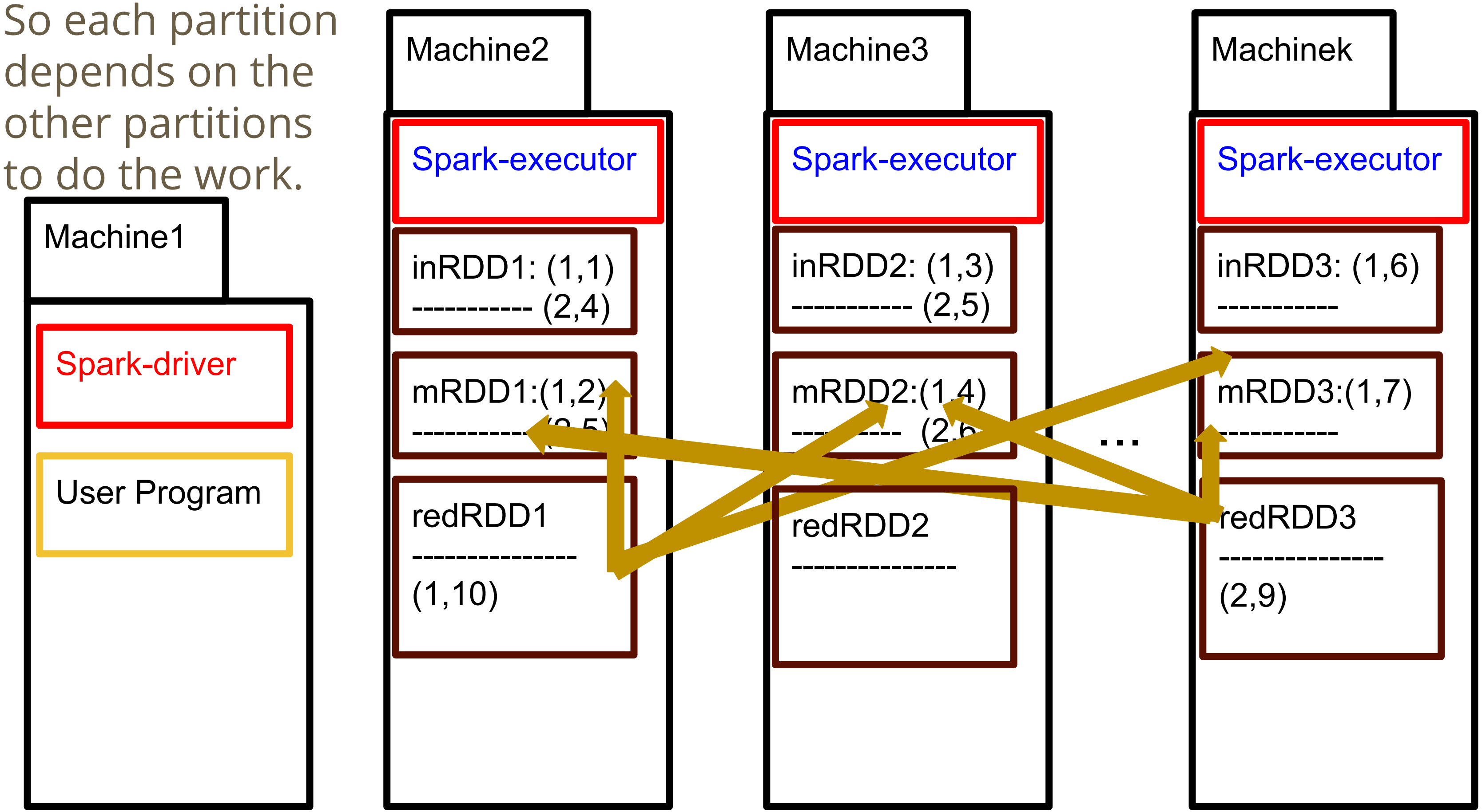
```
inputRDD  = sc.parallelize([(1,1), (2,4), (1,3), (2,5), (1,6)])
mapRDD = inputRDD.map( lambda elem : elem[1] + 1 )
redRDD = mapRDD.reduceByKey( lambda x, y: x + y )
solRDD = mapRDD.filter( lambda elem : elem[1] > 9 )
```

Lineage: Narrow and Wide Transformations



- Narrow vs Wide dependencies:

Second, the transformation reduceByKey shuffles the data among the partitions. So each partition depends on the other partitions to do the work.



Machine1

Spark-driver

User Program

Machine2

Spark-executor

inRDD1: (1,1)
----------- (2,4)

mRDD1:(1,2)
----------- (2,5)

redRDD1

(1,10)

Machine3

Spark-executor

inRDD2: (1,3)
----------- (2,5)

mRDD2:(1,4)
----------- (2,6)

redRDD2

…

Machinek

Spark-executor

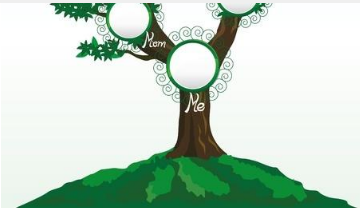
inRDD3: (1,6)
----

mRDD3:(1,7)
----

redRDD3

(2,9)

Lineage: Narrow and Wide Transformations

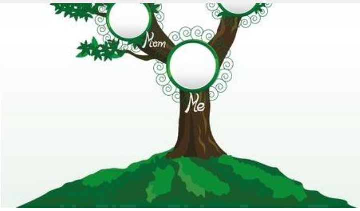


- Narrow vs Wide dependencies:

Third, the transformation filter can be done on its own by each partition.

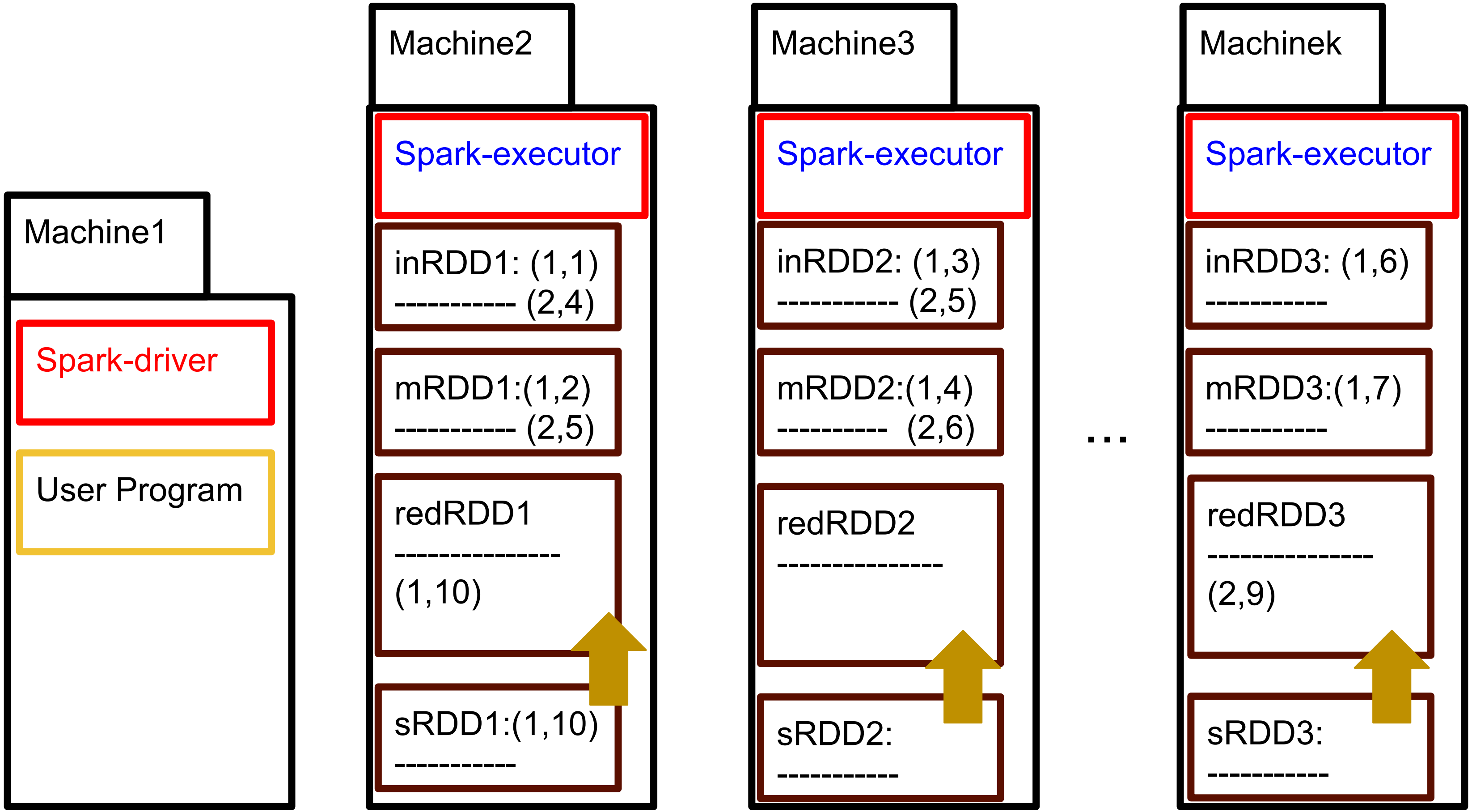
```
inputRDD  = sc.parallelize([(1,1), (2,4), (1,3), (2,5), (1,6)])
mapRDD = inputRDD.map( lambda elem : elem[1] + 1 )
redRDD = mapRDD.reduceByKey( lambda x, y: x + y )
solRDD = mapRDD.filter( lambda elem : elem[1] > 9 )
```

Lineage: Narrow and Wide Transformations



- Narrow vs Wide dependencies:

Third, the transformation filter can be done on its own by each partition.



Machine1
Spark-driver
User Program

Machine2
Spark-executor
inRDD1: (1,1)
----------- (2,4)
mRDD1:(1,2)
----------- (2,5)
redRDD1

(1,10)
sRDD1:(1,10)

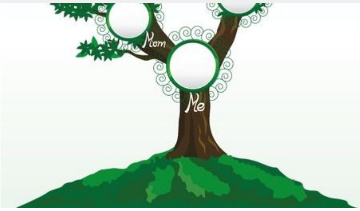
Machine3
Spark-executor
inRDD2: (1,3)
----------- (2,5)
mRDD2:(1,4)
---------- (2,6)
redRDD2
----
sRDD2:

…

Machinek
Spark-executor
inRDD3: (1,6)
----
mRDD3:(1,7)
----
redRDD3

(2,9)
sRDD3:

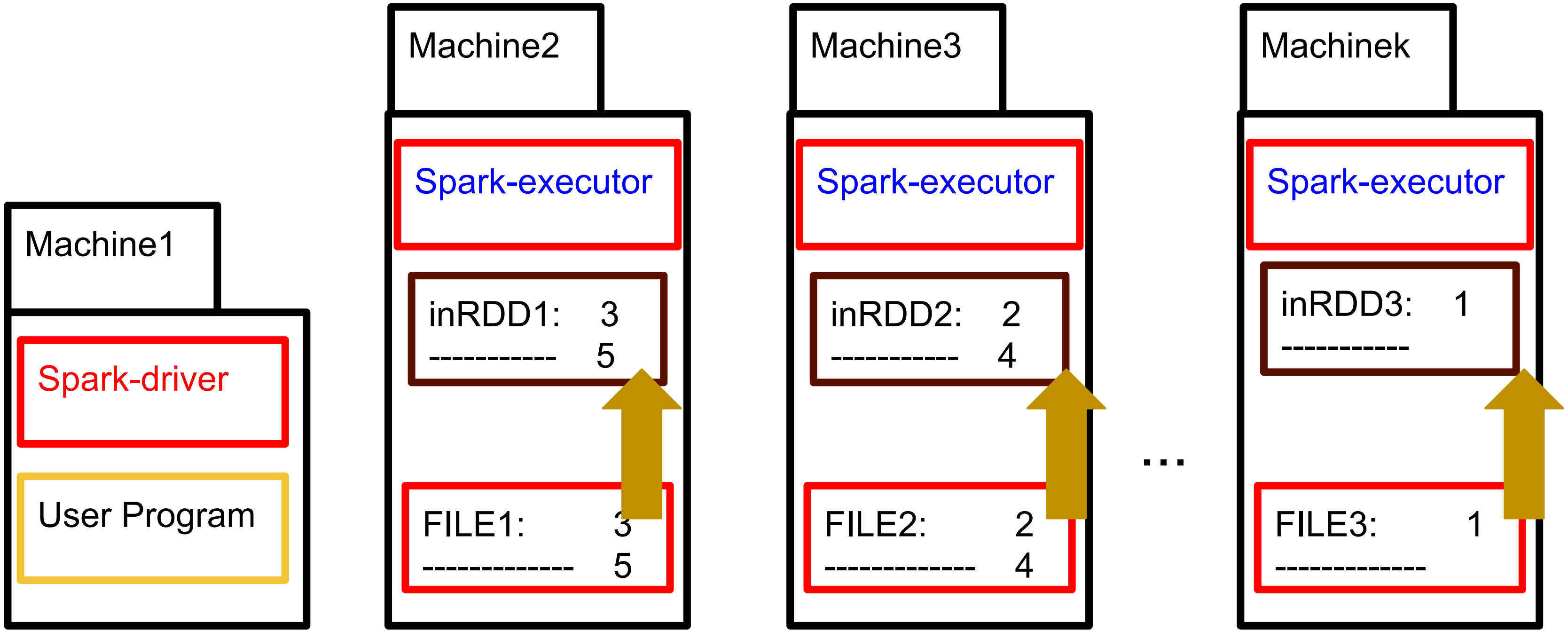
Lineage: Narrow and Wide Transformations



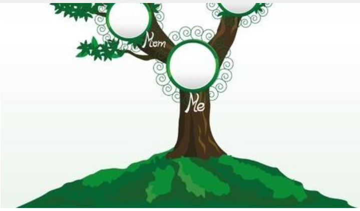
The same rationale of narrow and wide dependencies we have studied for **creation** and **transformations** apply as well to **actions**.

➢ In this example, saveAsTextFile acts as a narrow dependency-based **action**:

```
inputRDD  = sc.parallelize([1, 2, 3, 4, 5])
inputRDD.saveAsTextFile(my_new_directory)
```



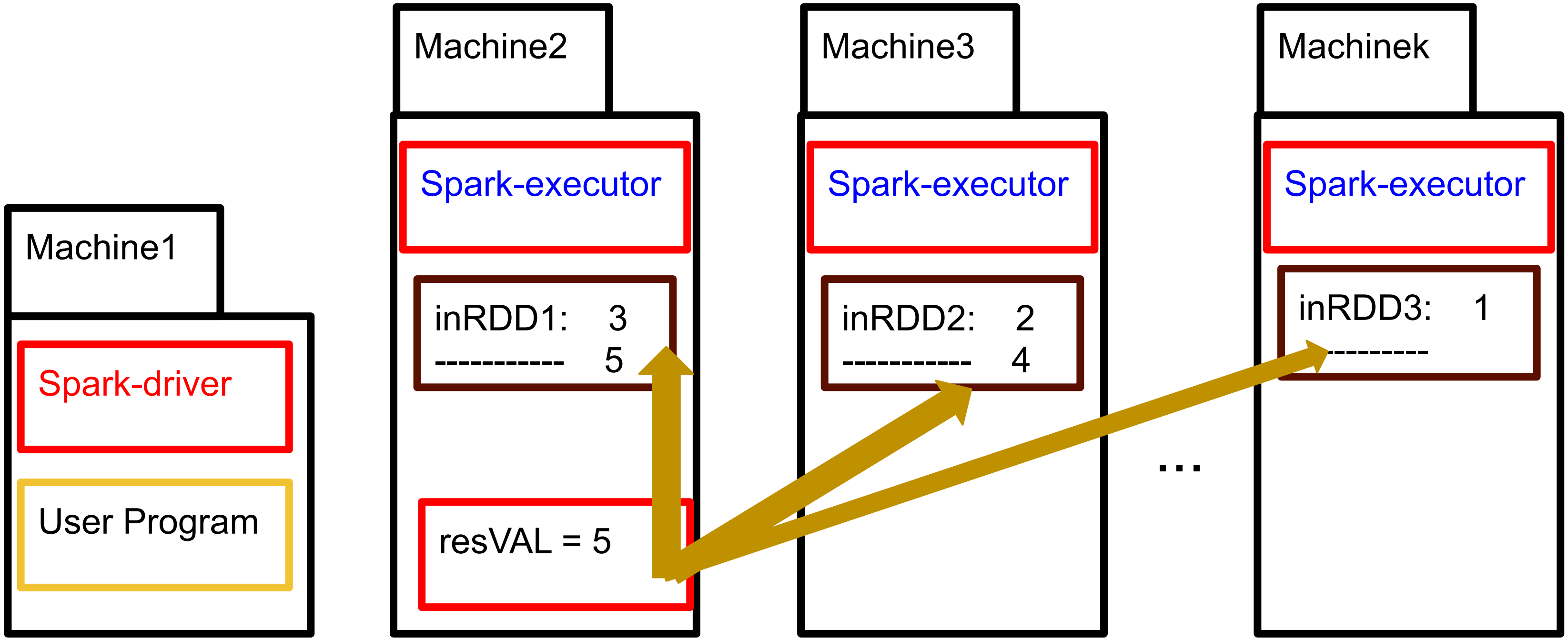
Lineage: Narrow and Wide Transformations



The same rationale of narrow and wide dependencies we have studied for **creation** and **transformations** apply as well to **actions**.

➢ In this example, count acts as a wide dependency-based **action**:

```
inputRDD  = sc.parallelize([1, 2, 3, 4, 5])
resVAL = inputRDD.count()
print(resVAL)
```



Machine1

Spark-driver

User Program

Machine2

Spark-executor

inRDD1: 3
----------- 5

resVAL = 5

Machine3

Spark-executor

inRDD2: 2
----------- 4

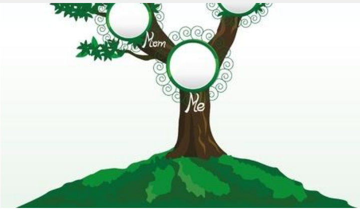
…

Machinek

Spark-executor

inRDD3: 1

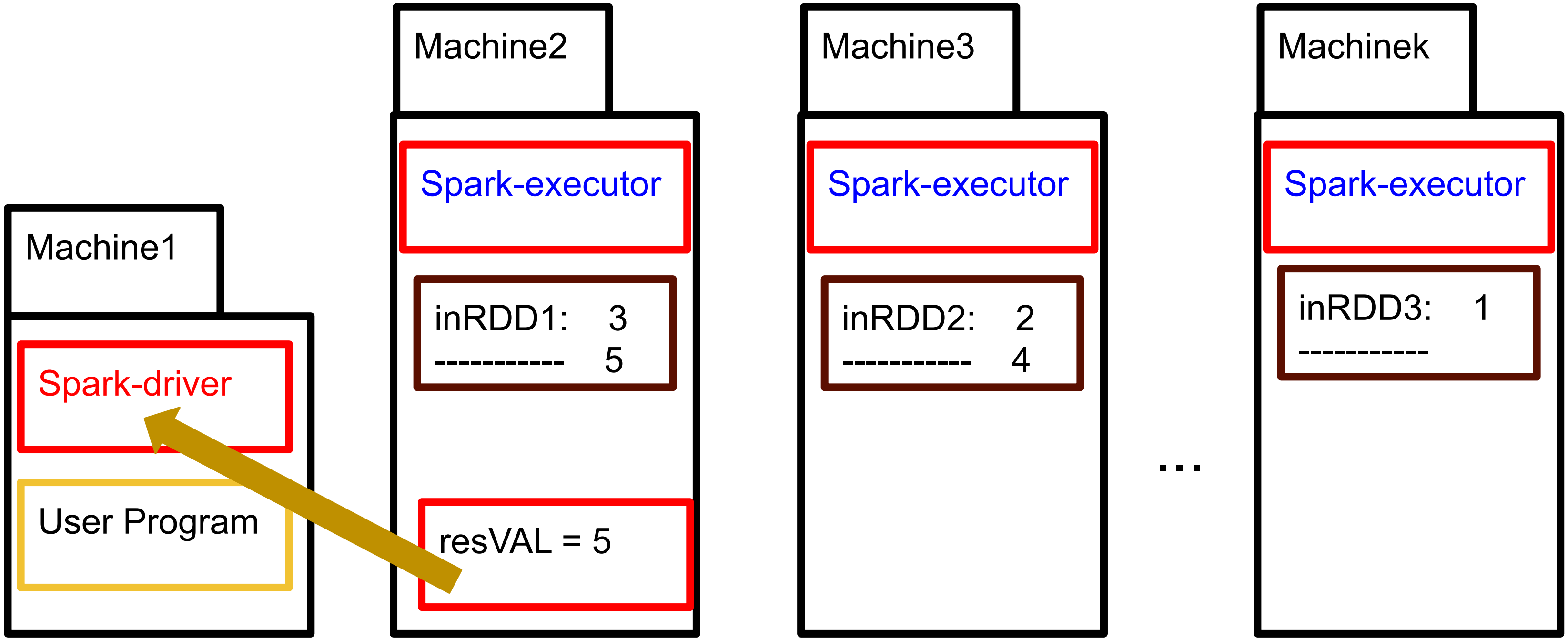
Lineage: Narrow and Wide Transformations



The same rationale of narrow and wide dependencies we have studied for **creation** and **transformations** apply as well to **actions**.

➢ In this example, count acts as a wide dependency-based **action**:

```
inputRDD  = sc.parallelize([1, 2, 3, 4, 5])
resVAL = inputRDD.count()
print(resVAL)
```



Outline

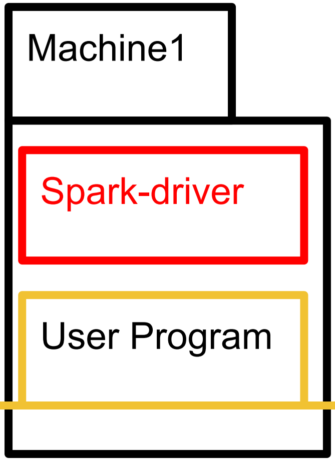
1. Setting the Context.
2. RDD Private Side: Partitions and Lineage.
   a. Internal Representation.
   b. Partitions.
   c. Lineage: Narrow and Wide Transformations.
   d. Lineage: Lazy evaluation.
   e. Lineage: Lazy evaluation and Persistance.
   f. Lineage: Fault tolerant.
3. Spark Application: Jobs, Stages and Tasks.

Lineage: Lazy Evaluation

Now, why do we claim lineage to be crucial for implementing lazy evaluation?

Lineage: Lazy Evaluation

Given a Spark User program P…



1. **Creator**: They create a new RDD from an existing collection or dataset.
2. **Mutator**: These operations are called **Transformations**.
   They take one or more RDDs and produce a new RDD.
3. **Persistent**: They keep an RDD permanently stored until the Spark program finishes.
4. **Observer**: These operations are called **Actions**.
   They return some property/info from an RDD without modifying it.

Lineage: Lazy Evaluation

Given a Spark User program P:

Any **cretor**, **transformation** and **persistent** operation is registered….

```
inputRDD  = sc.parallelize( [ 1, 2, 3, 4, 5 ] )
mappedRDD  = inputRDD.map( lambda elem : elem + 1 )
filteredRDD  = mappedRDD.filter( lambda elem : elem > 3 )
resVAL = filteredRDD.count()
```

Lineage: Lazy Evaluation

Given a Spark User program P:

Any **cretor**, **transformation** and **persistent** operation is registered….
but not actually computed!

```
inputRDD   = sc.parallelize( [ 1, 2, 3, 4, 5 ] )
mappedRDD   = inputRDD.map( lambda elem : elem + 1 )
filteredRDD   = mappedRDD.filter( lambda elem : elem > 3 )
resVAL = filteredRDD.count()
```

Lineage: Lazy Evaluation

Given a Spark User program P:

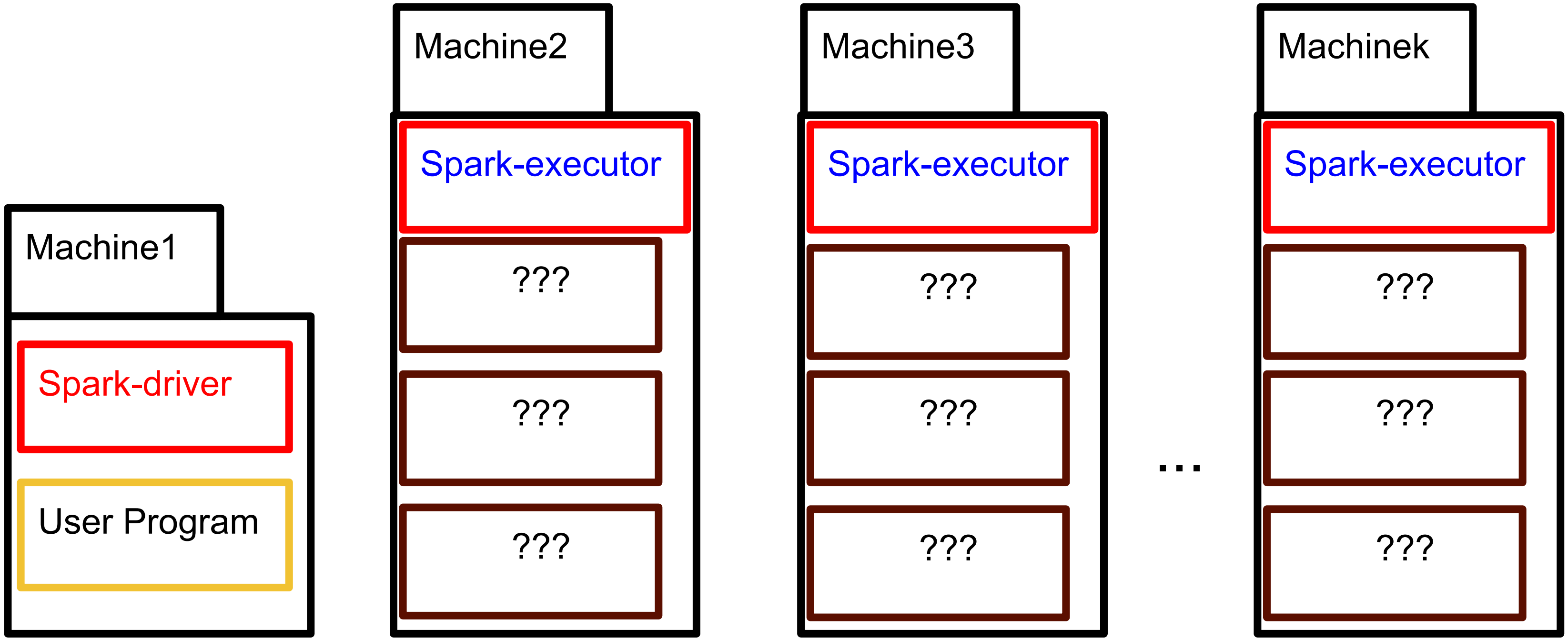
Any **cretor**, **transformation** and **persistent** operation is registered....
but not actually computed!
until an **action** comes to the rescue!

```
inputRDD   = sc.parallelize( [ 1, 2, 3, 4, 5 ] )
mappedRDD   = inputRDD.map( lambda elem : elem + 1 )
filteredRDD   = mappedRDD.filter( lambda elem : elem > 3 )
resVAL = filteredRDD.count()
```

Lineage: Lazy Evaluation

So, what is registered then?

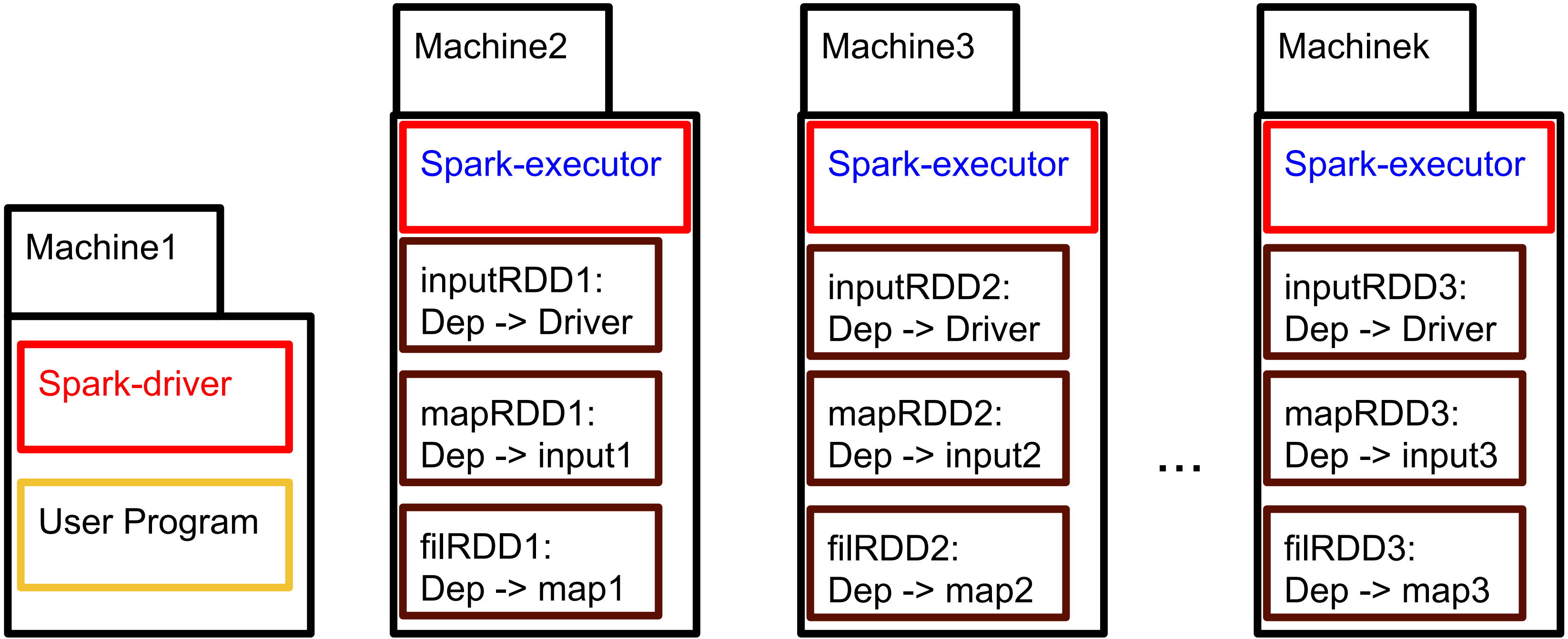
```
inputRDD   = sc.parallelize( [ 1, 2, 3, 4, 5 ] )
mappedRDD   = inputRDD.map( lambda elem : elem + 1 )
filteredRDD   = mappedRDD.filter( lambda elem : elem > 3 )
resVAL = filteredRDD.count()
```



Lineage: Lazy Evaluation

So, what is registered then?
The lineage!

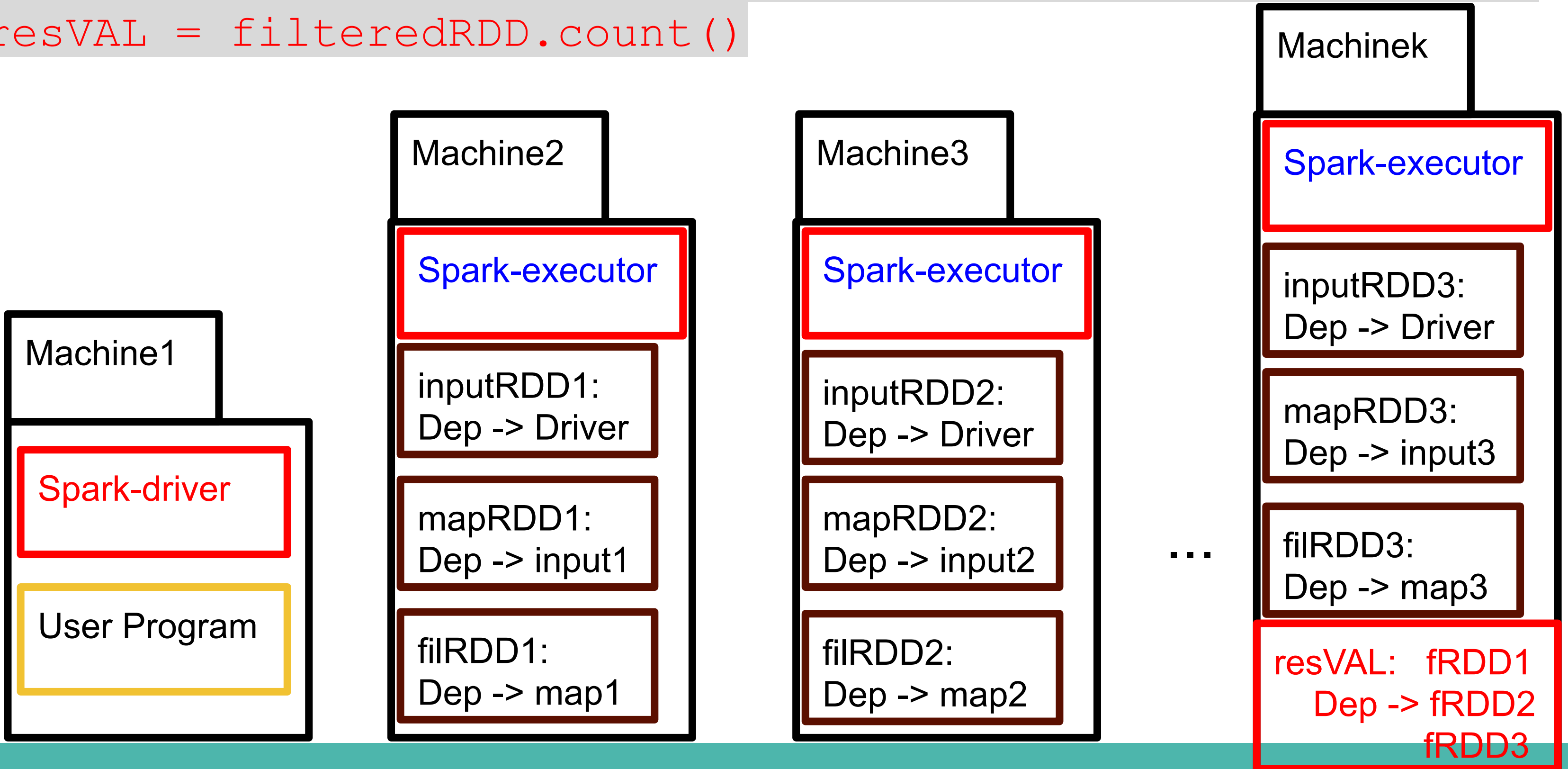
```
inputRDD  = sc.parallelize( [ 1, 2, 3, 4, 5 ] )
mappedRDD  = inputRDD.map( lambda elem : elem + 1 )
filteredRDD  = mappedRDD.filter( lambda elem : elem > 3 )
resVAL = filteredRDD.count()
```



Lineage: Lazy Evaluation

And when an **action** takes place, computation is triggered by tracing the lineage backwards.

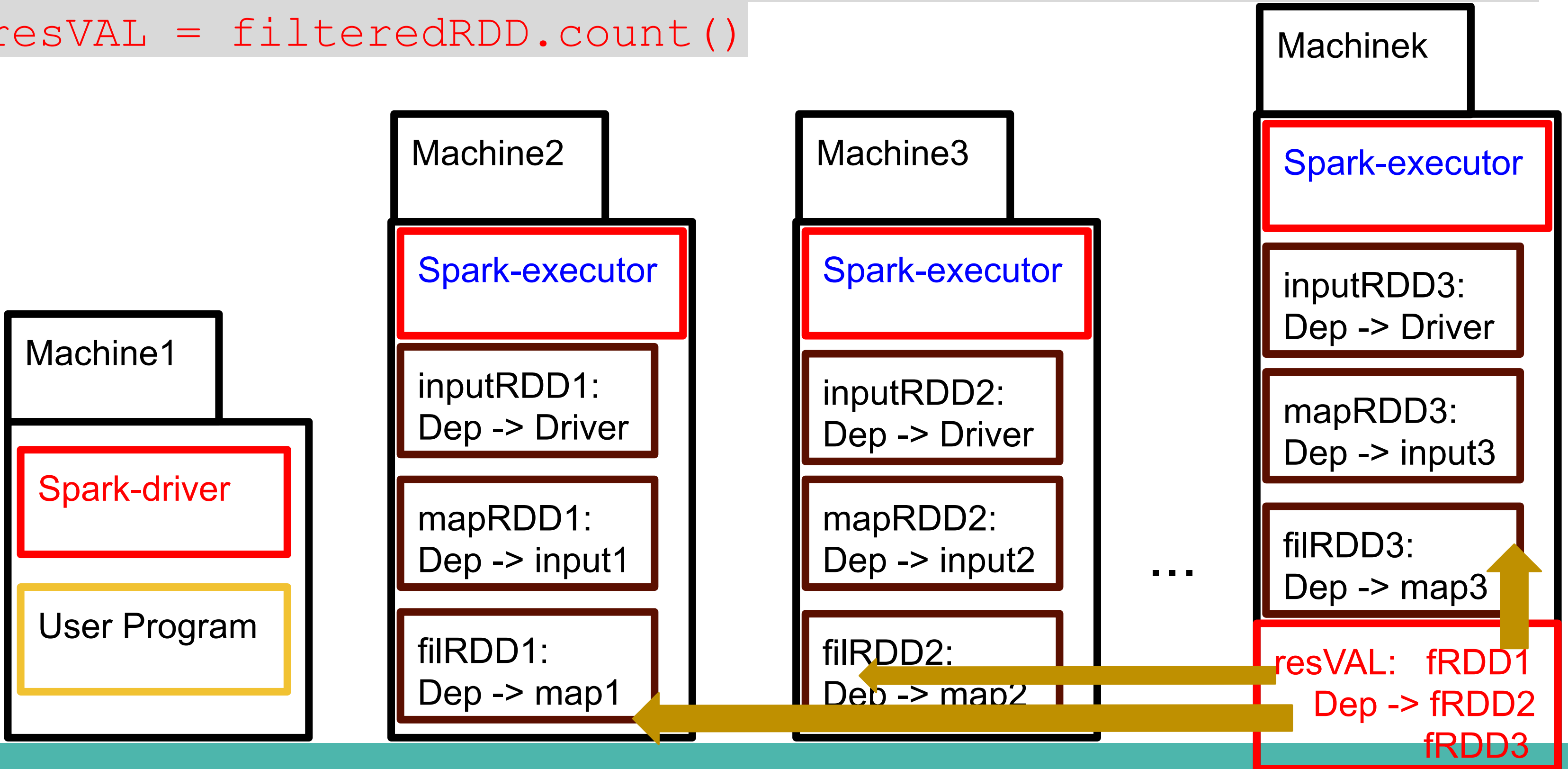
```
inputRDD  = sc.parallelize( [ 1, 2, 3, 4, 5 ] )
mappedRDD  = inputRDD.map( lambda elem : elem + 1 )
filteredRDD  = mappedRDD.filter( lambda elem : elem > 3 )
resVAL = filteredRDD.count()
```



Lineage: Lazy Evaluation

Who do I depend on?

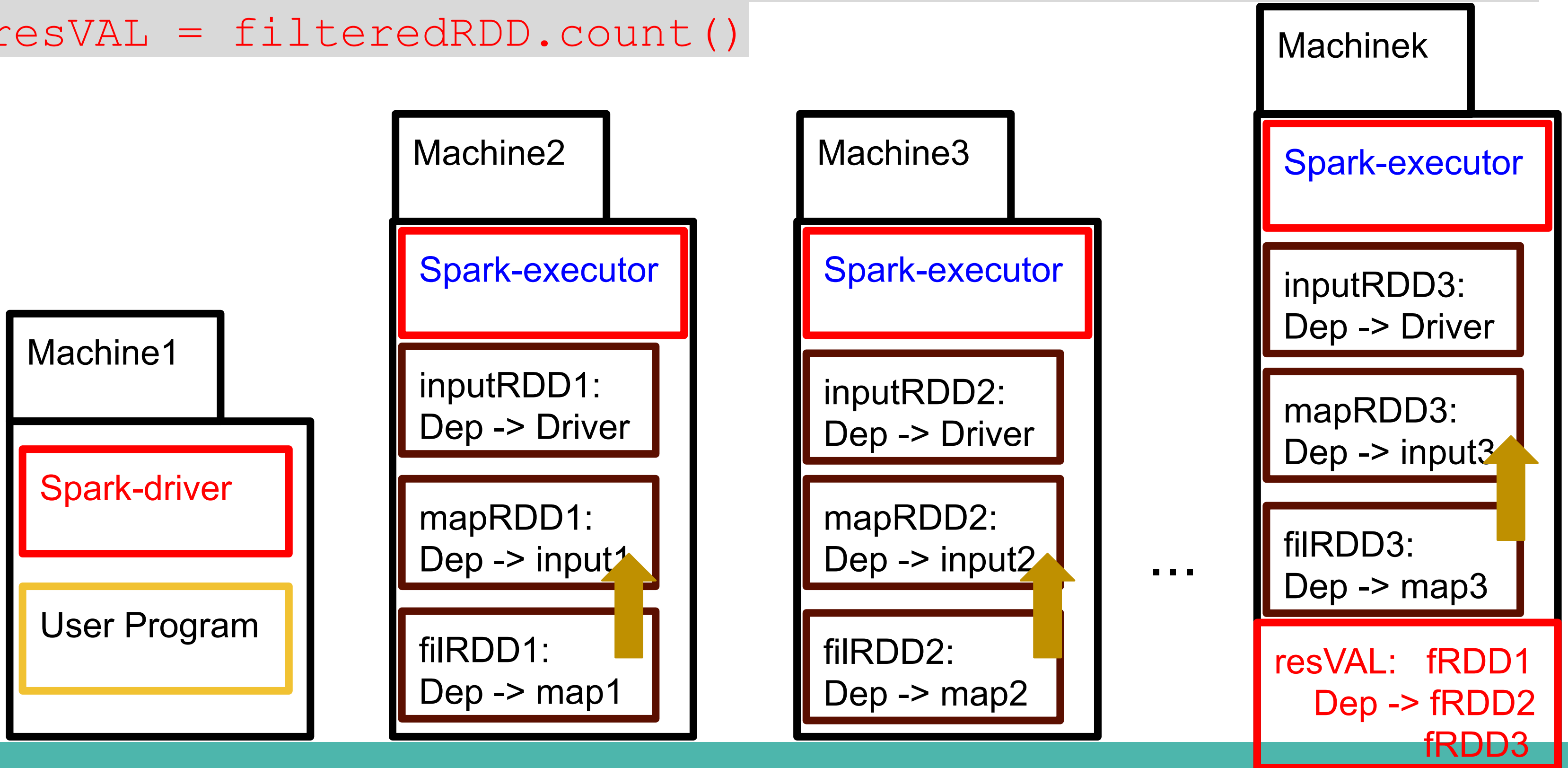
```
inputRDD  = sc.parallelize( [ 1, 2, 3, 4, 5 ] )
mappedRDD  = inputRDD.map( lambda elem : elem + 1 )
filteredRDD  = mappedRDD.filter( lambda elem : elem > 3 )
resVAL = filteredRDD.count()
```



Lineage: Lazy Evaluation

And, likewise, who do these RDD partitions depend on?

```
inputRDD  = sc.parallelize( [ 1, 2, 3, 4, 5 ] )
mappedRDD  = inputRDD.map( lambda elem : elem + 1 )
filteredRDD  = mappedRDD.filter( lambda elem : elem > 3 )
resVAL = filteredRDD.count()
```



Machine1

Spark-driver

User Program

Machine2

Spark-executor

inputRDD1:
Dep -> Driver

mapRDD1:
Dep -> input1

filRDD1:
Dep -> map1

Machine3

Spark-executor

inputRDD2:
Dep -> Driver

mapRDD2:
Dep -> input2

filRDD2:
Dep -> map2

…

Machinek

Spark-executor

inputRDD3:
Dep -> Driver

mapRDD3:
Dep -> input3

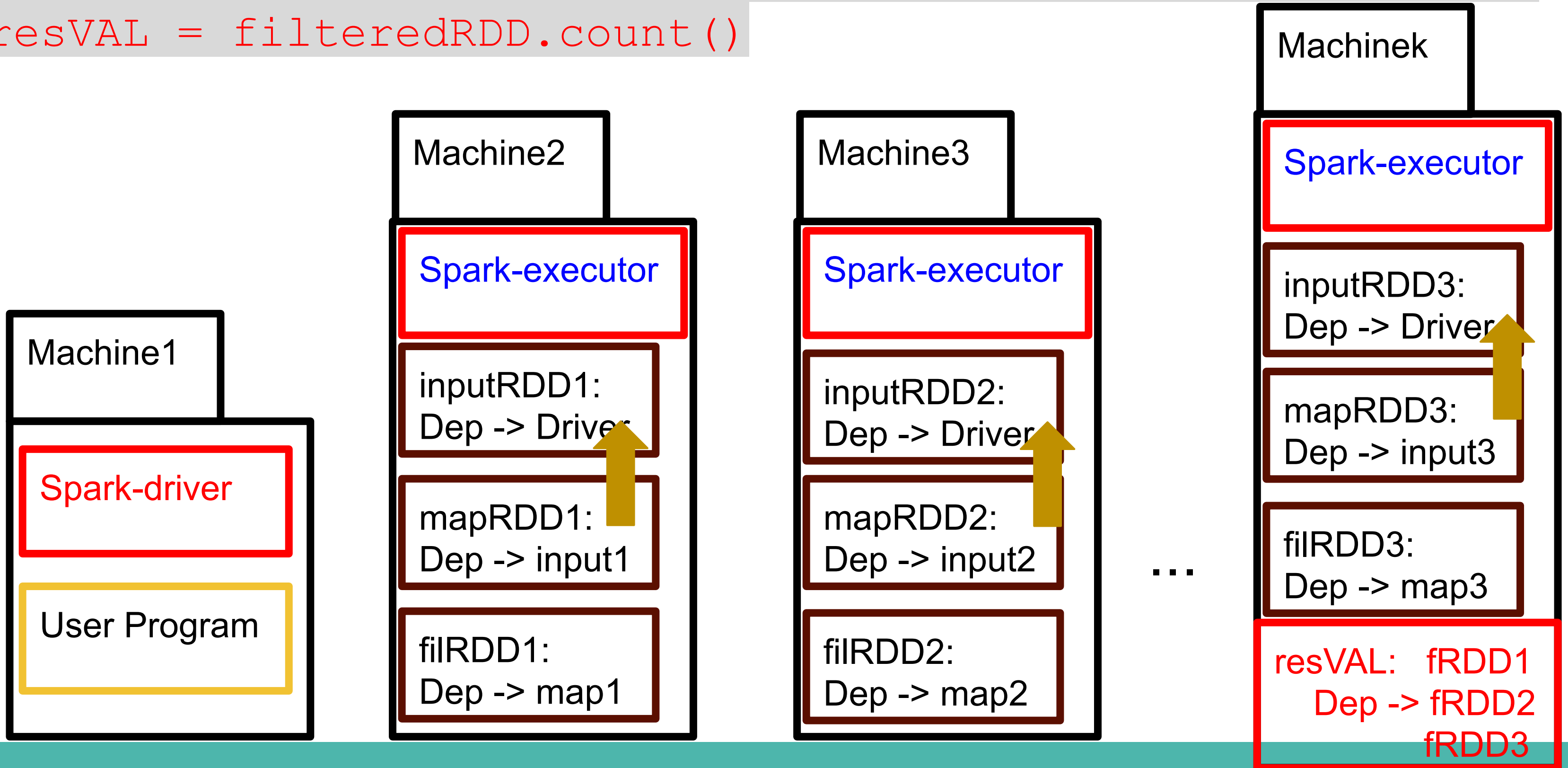
filRDD3:
Dep -> map3

resVAL: fRDD1
Dep -> fRDD2
fRDD3

Lineage: Lazy Evaluation

And so on and so on…

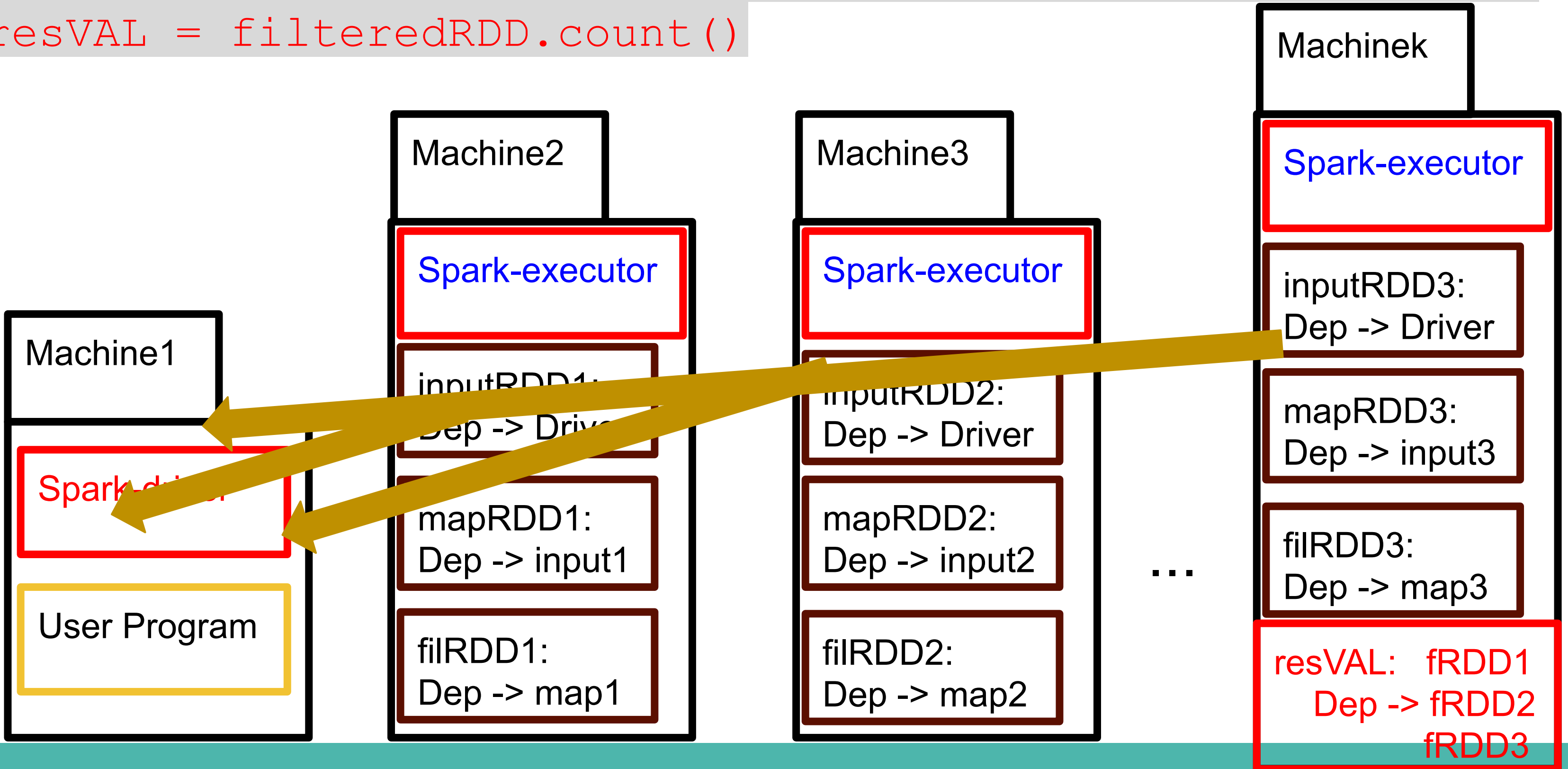
```
inputRDD  = sc.parallelize( [ 1, 2, 3, 4, 5 ] )
mappedRDD  = inputRDD.map( lambda elem : elem + 1 )
filteredRDD  = mappedRDD.filter( lambda elem : elem > 3 )
resVAL = filteredRDD.count()
```



Lineage: Lazy Evaluation

And so on and so on…

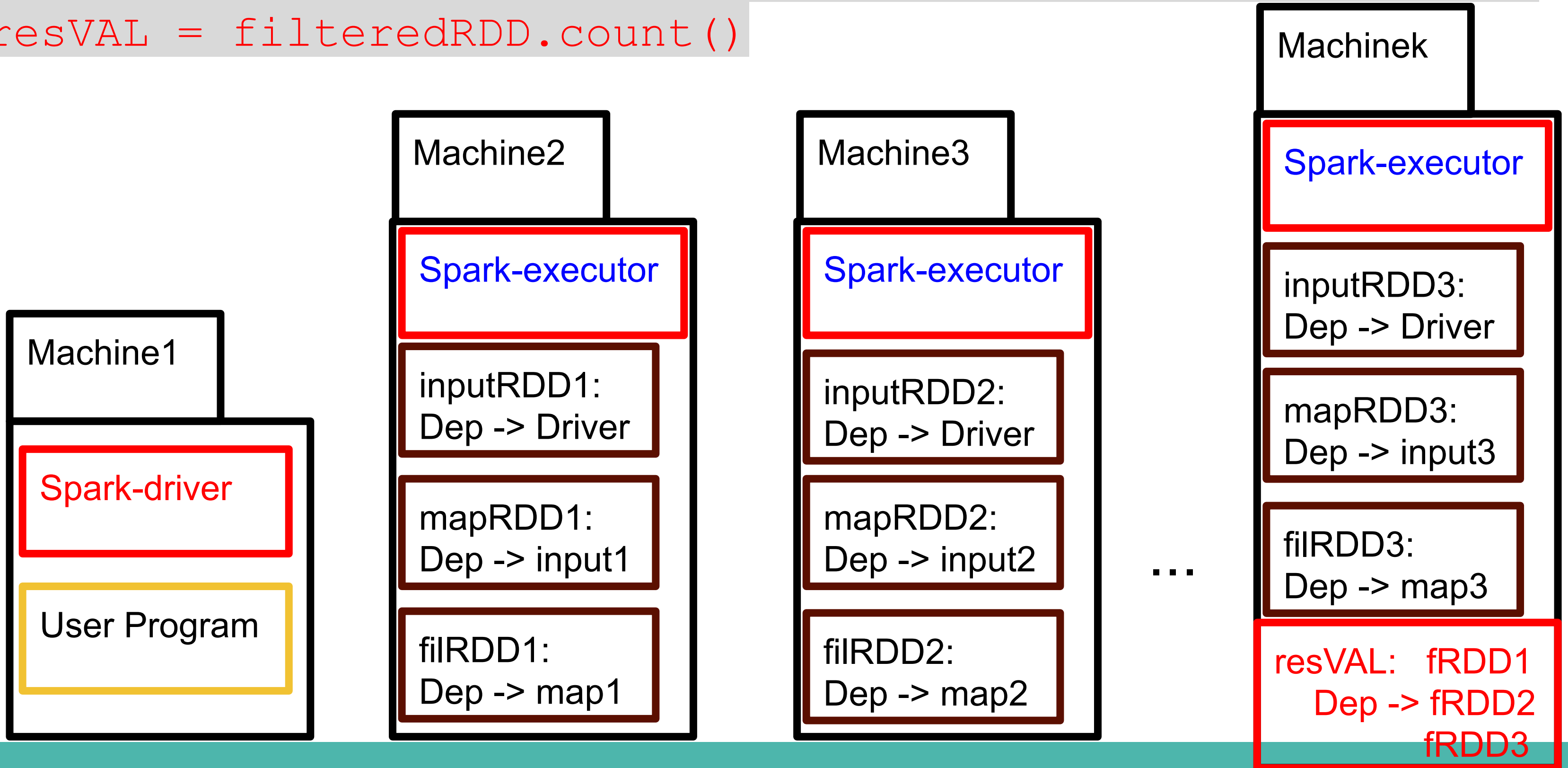
```
inputRDD  = sc.parallelize( [ 1, 2, 3, 4, 5 ] )
mappedRDD  = inputRDD.map( lambda elem : elem + 1 )
filteredRDD  = mappedRDD.filter( lambda elem : elem > 3 )
resVAL = filteredRDD.count()
```



Lineage: Lazy Evaluation

And now that I know the full lineage, computation can start lazily.

```
inputRDD   = sc.parallelize( [ 1, 2, 3, 4, 5 ] )
mappedRDD   = inputRDD.map( lambda elem : elem + 1 )
filteredRDD   = mappedRDD.filter( lambda elem : elem > 3 )
resVAL = filteredRDD.count()
```



Machine1
Spark-driver
User Program

Machine2
Spark-executor
inputRDD1: Dep -> Driver
mapRDD1: Dep -> input1
filRDD1: Dep -> map1

Machine3
Spark-executor
inputRDD2: Dep -> Driver
mapRDD2: Dep -> input2
filRDD2: Dep -> map2

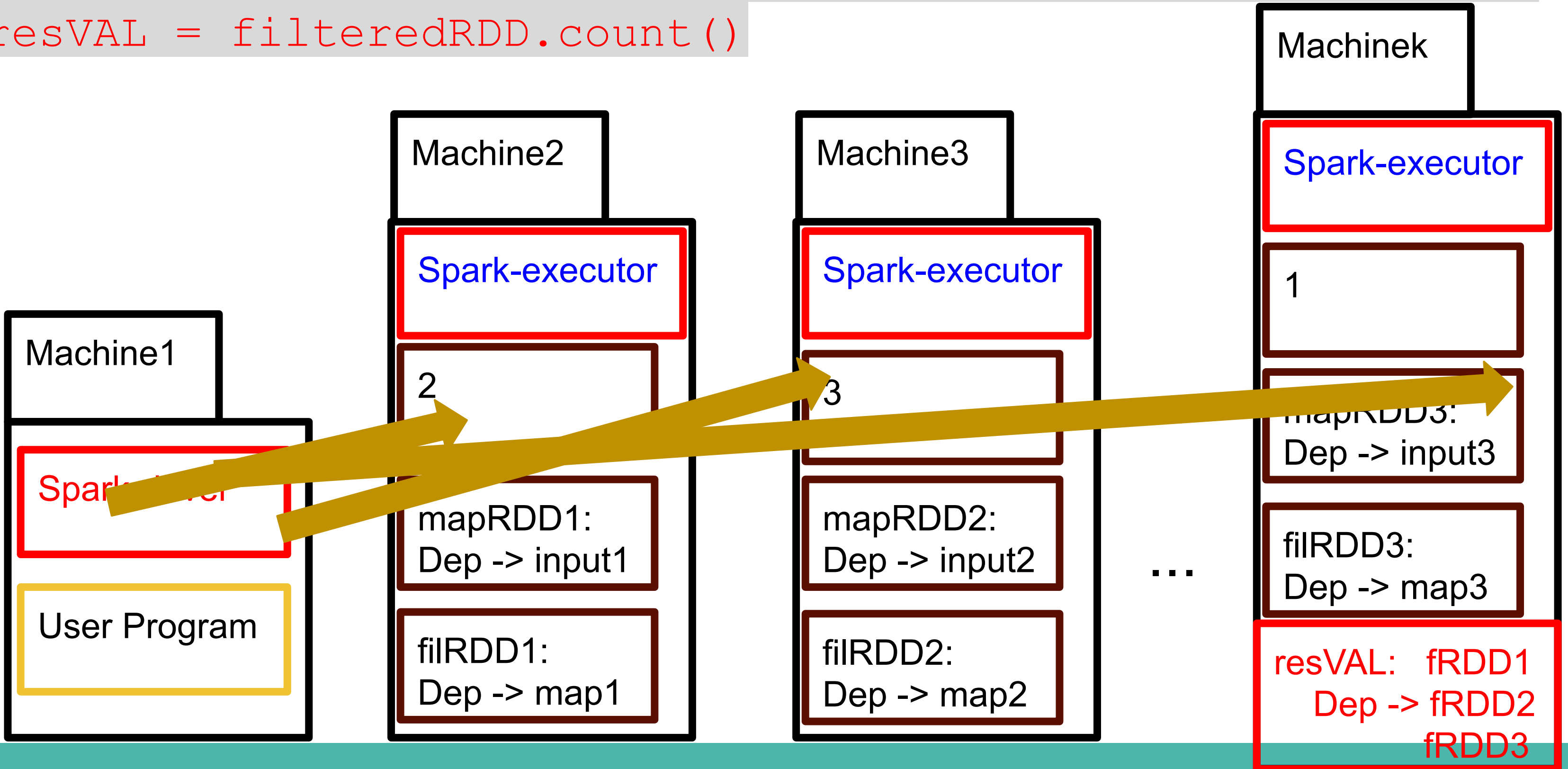
…

Machinek
Spark-executor
inputRDD3: Dep -> Driver
mapRDD3: Dep -> input3
filRDD3: Dep -> map3
resVAL: fRDD1 Dep -> fRDD2 fRDD3

Lineage: Lazy Evaluation

In this case, going forward!

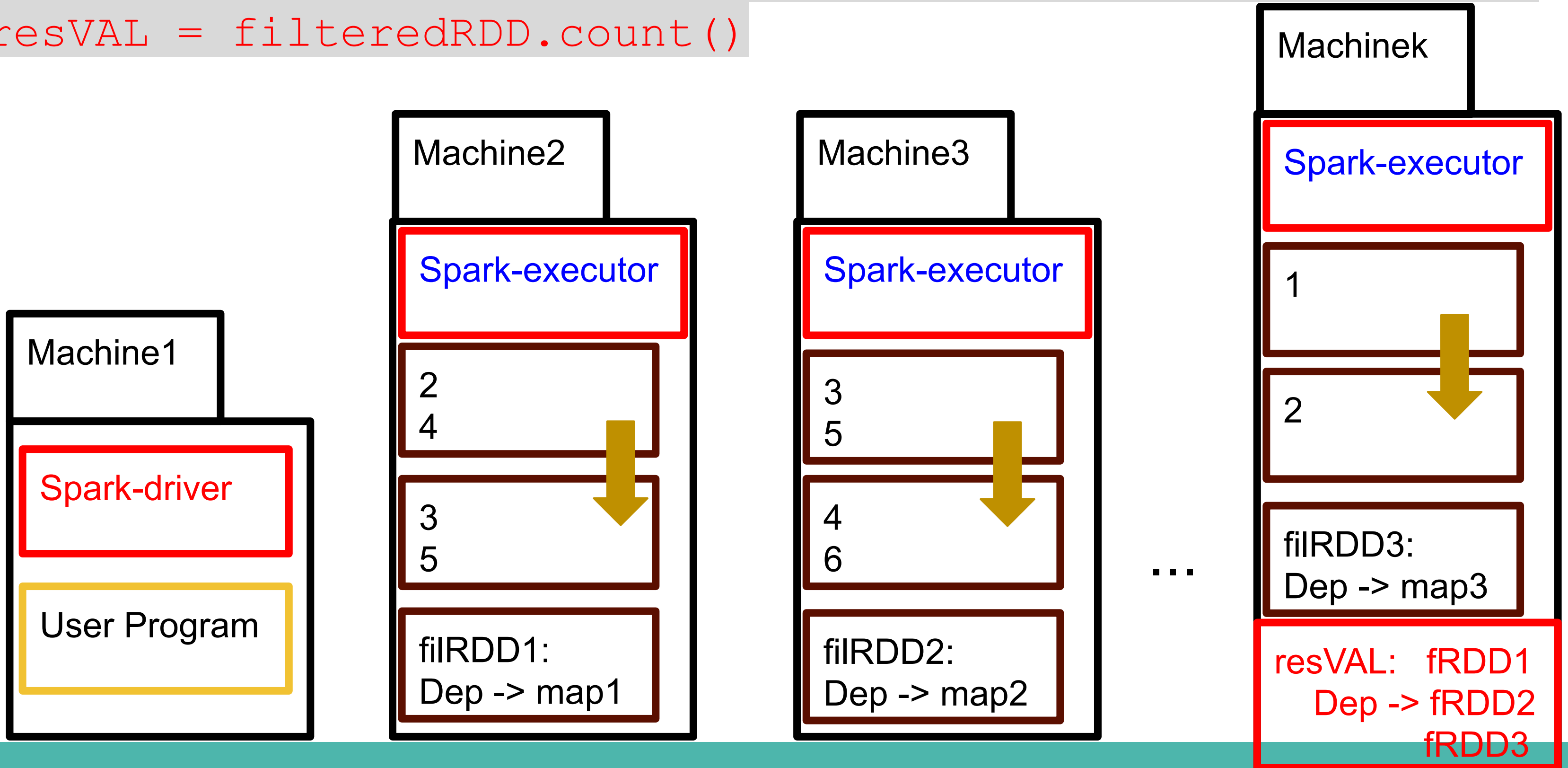
```
inputRDD   = sc.parallelize( [ 1, 2, 3, 4, 5 ] )
mappedRDD  = inputRDD.map( lambda elem : elem + 1 )
filteredRDD  = mappedRDD.filter( lambda elem : elem > 3 )
resVAL = filteredRDD.count()
```



Lineage: Lazy Evaluation

In this case, going forward!

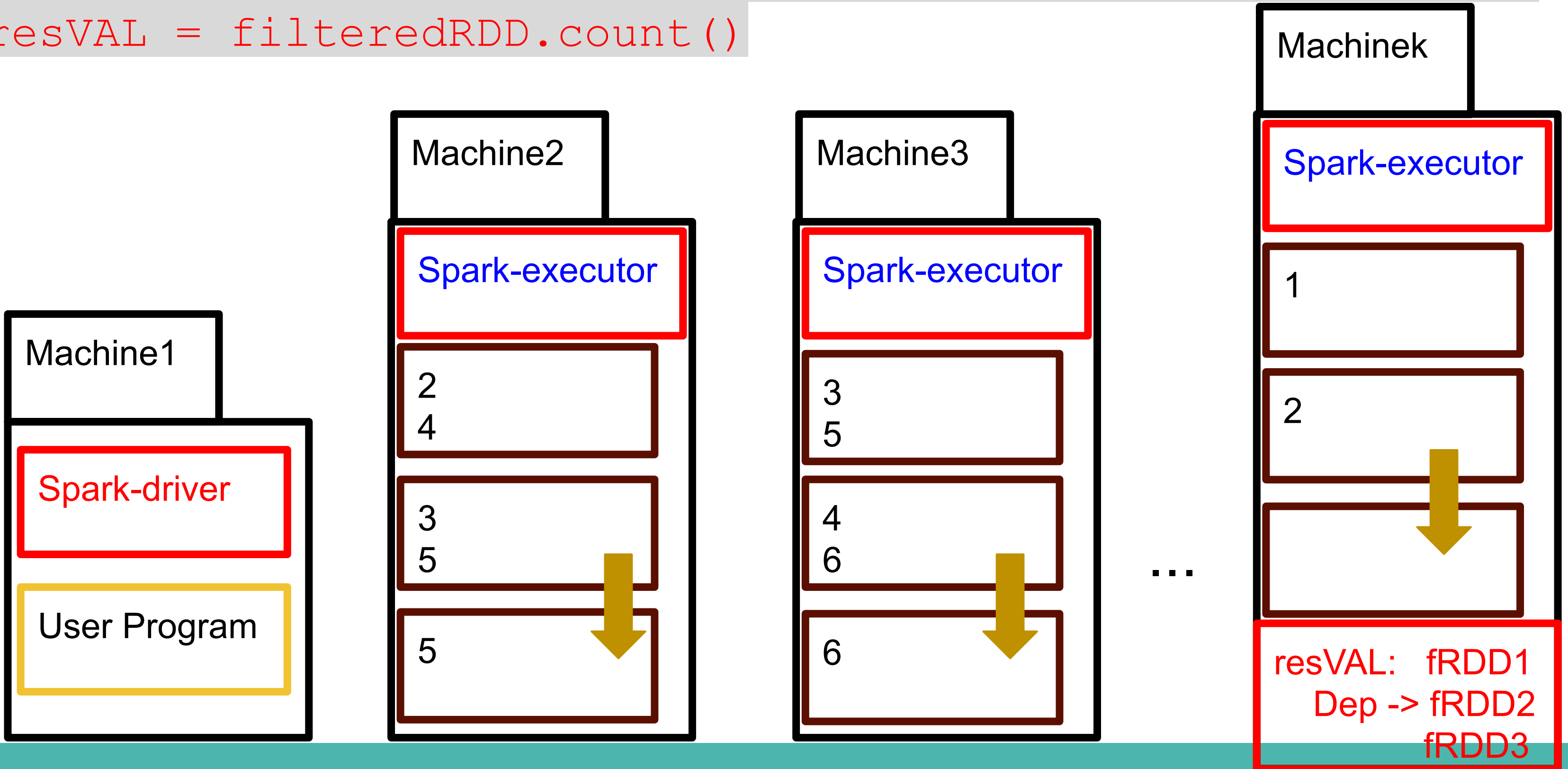
```
inputRDD   = sc.parallelize( [ 1, 2, 3, 4, 5 ] )
mappedRDD  = inputRDD.map( lambda elem : elem + 1 )
filteredRDD  = mappedRDD.filter( lambda elem : elem > 3 )
resVAL = filteredRDD.count()
```



Lineage: Lazy Evaluation

In this case, going forward!

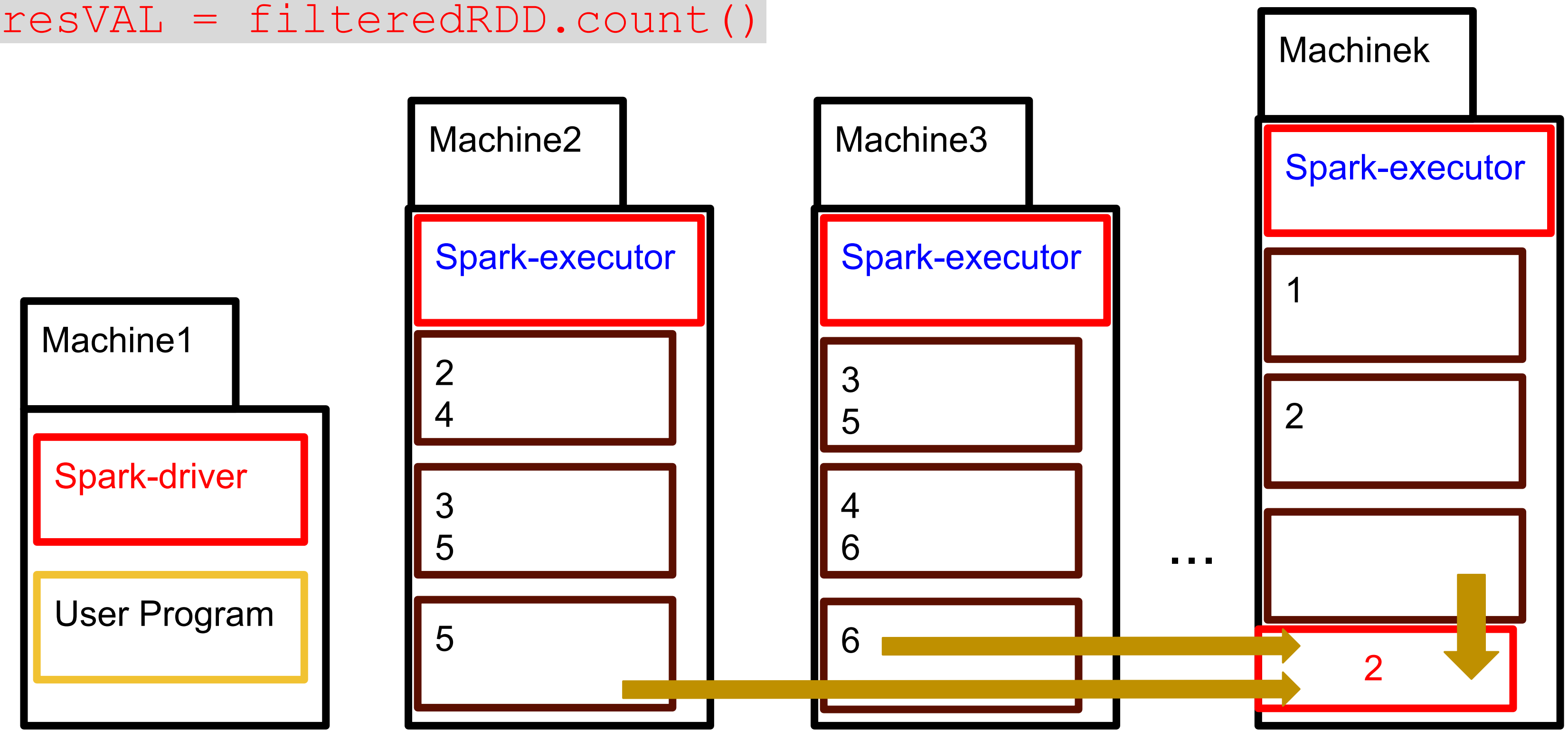
```
inputRDD  = sc.parallelize( [ 1, 2, 3, 4, 5 ] )
mappedRDD  = inputRDD.map( lambda elem : elem + 1 )
filteredRDD  = mappedRDD.filter( lambda elem : elem > 3 )
resVAL = filteredRDD.count()
```



Lineage: Lazy Evaluation

In this case, going forward!

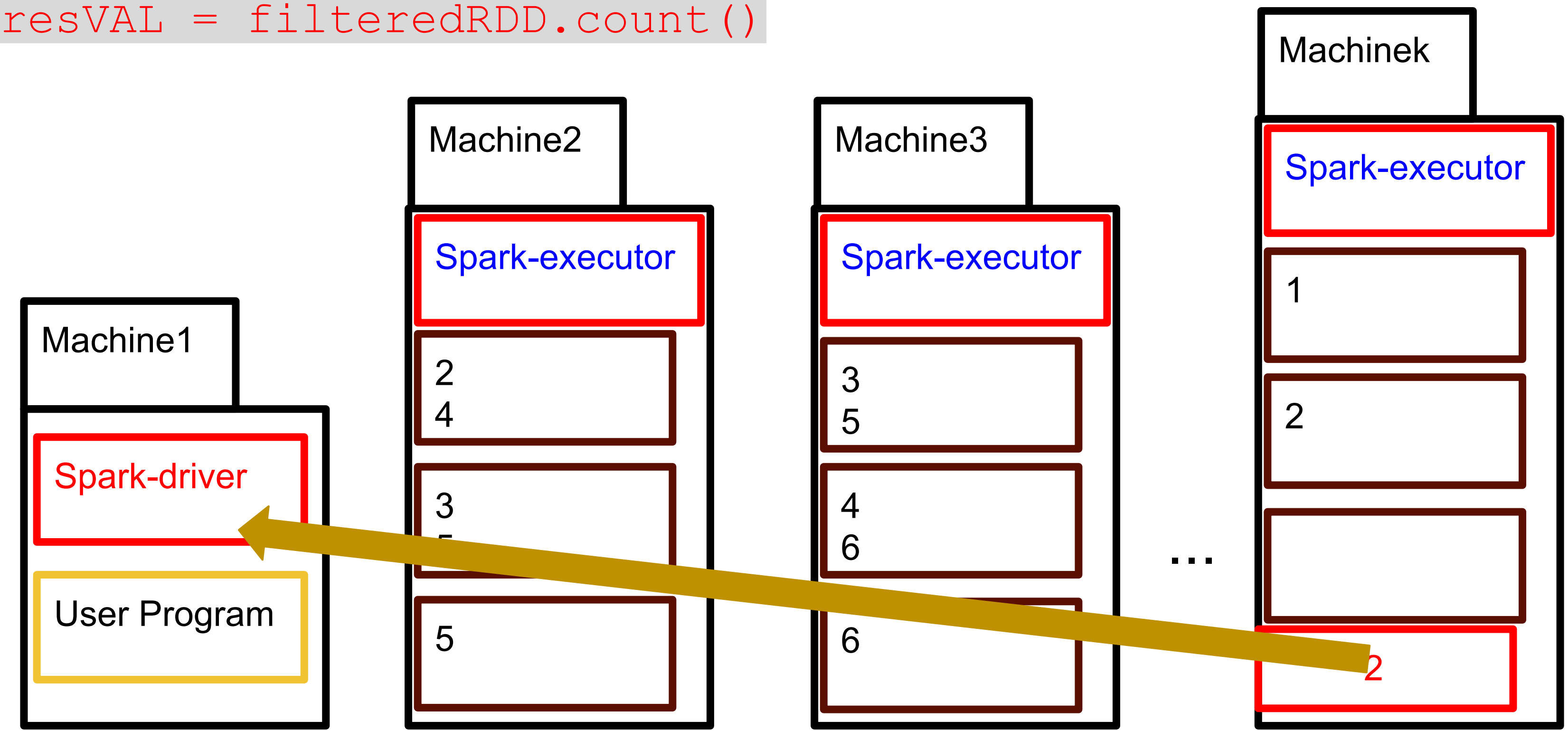
```
inputRDD   = sc.parallelize( [ 1, 2, 3, 4, 5 ] )
mappedRDD   = inputRDD.map( lambda elem : elem + 1 )
filteredRDD   = mappedRDD.filter( lambda elem : elem > 3 )
resVAL = filteredRDD.count()
```



Lineage: Lazy Evaluation

In this case, going forward!

```
inputRDD   = sc.parallelize( [ 1, 2, 3, 4, 5 ] )
mappedRDD   = inputRDD.map( lambda elem : elem + 1 )
filteredRDD   = mappedRDD.filter( lambda elem : elem > 3 )
resVAL = filteredRDD.count()
```



Outline

1. Setting the Context.
2. RDD Private Side: Partitions and Lineage.
   a. Internal Representation.
   b. Partitions.
   c. Lineage: Narrow and Wide Transformations.
   d. Lineage: Lazy evaluation.
   e. Lineage: Lazy evaluation and Persistance.
   f. Lineage: Fault tolerant.
3. Spark Application: Jobs, Stages and Tasks.

Lineage: Lazy Evaluation and Persistance

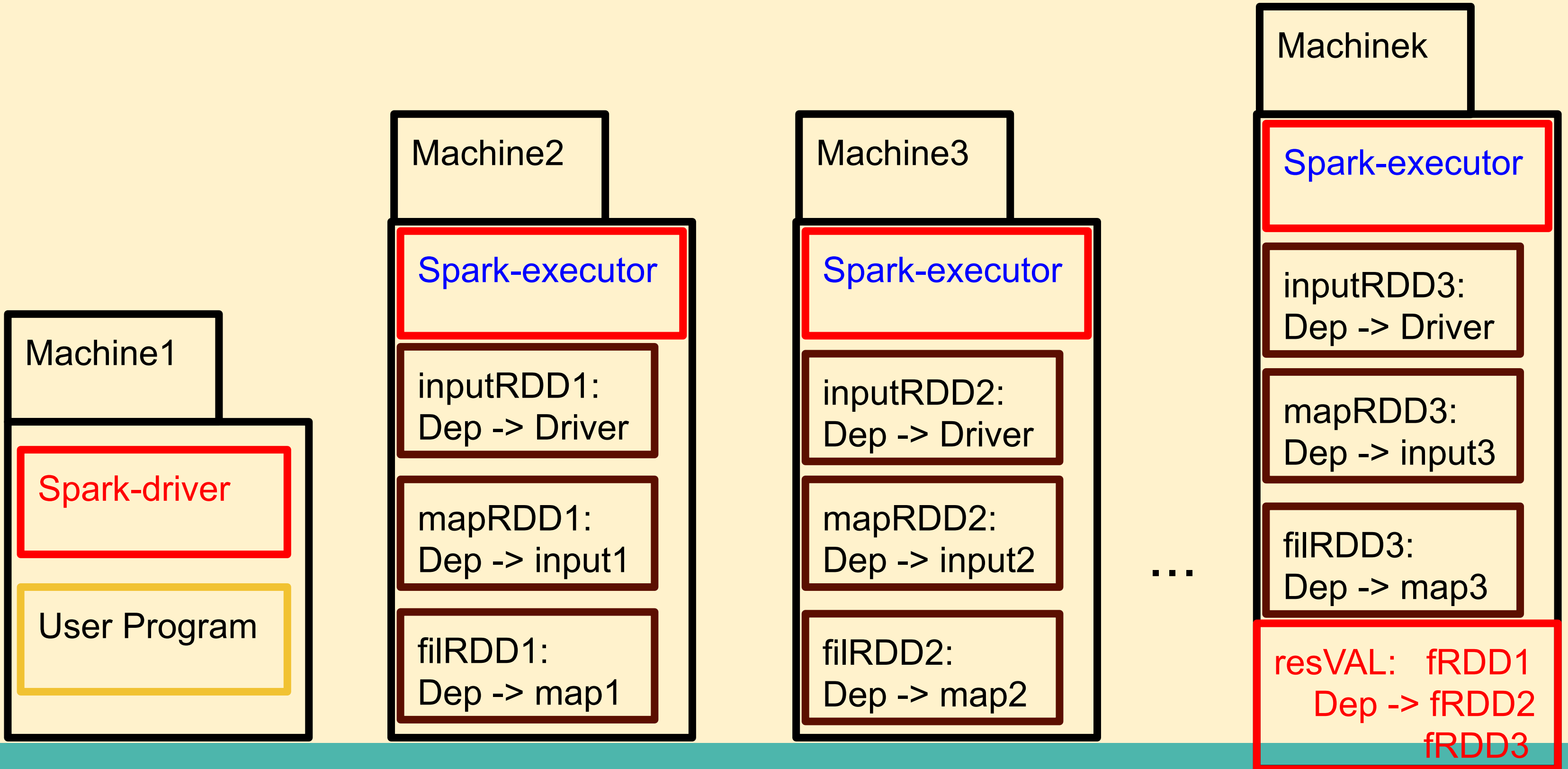
Now that we understand how lineage allows lazy evaluation to happen, it is time to correct one mistake from the previous slides:

Actually, RDD partitions are not computed and kept in memory!

Lineage: Lazy Evaluation and Persistance

…this is not exactly what happens…

```
inputRDD  = sc.parallelize( [ 1, 2, 3, 4, 5 ] )
mappedRDD  = inputRDD.map( lambda elem : elem + 1 )
filteredRDD  = mappedRDD.filter( lambda elem : elem > 3 )
resVAL = filteredRDD.count()
```



Machine1

Spark-driver

User Program

Machine2

Spark-executor

inputRDD1: Dep -> Driver

mapRDD1: Dep -> input1

filRDD1: Dep -> map1

Machine3

Spark-executor

inputRDD2: Dep -> Driver

mapRDD2: Dep -> input2

filRDD2: Dep -> map2

…

Machinek

Spark-executor

inputRDD3: Dep -> Driver

mapRDD3: Dep -> input3

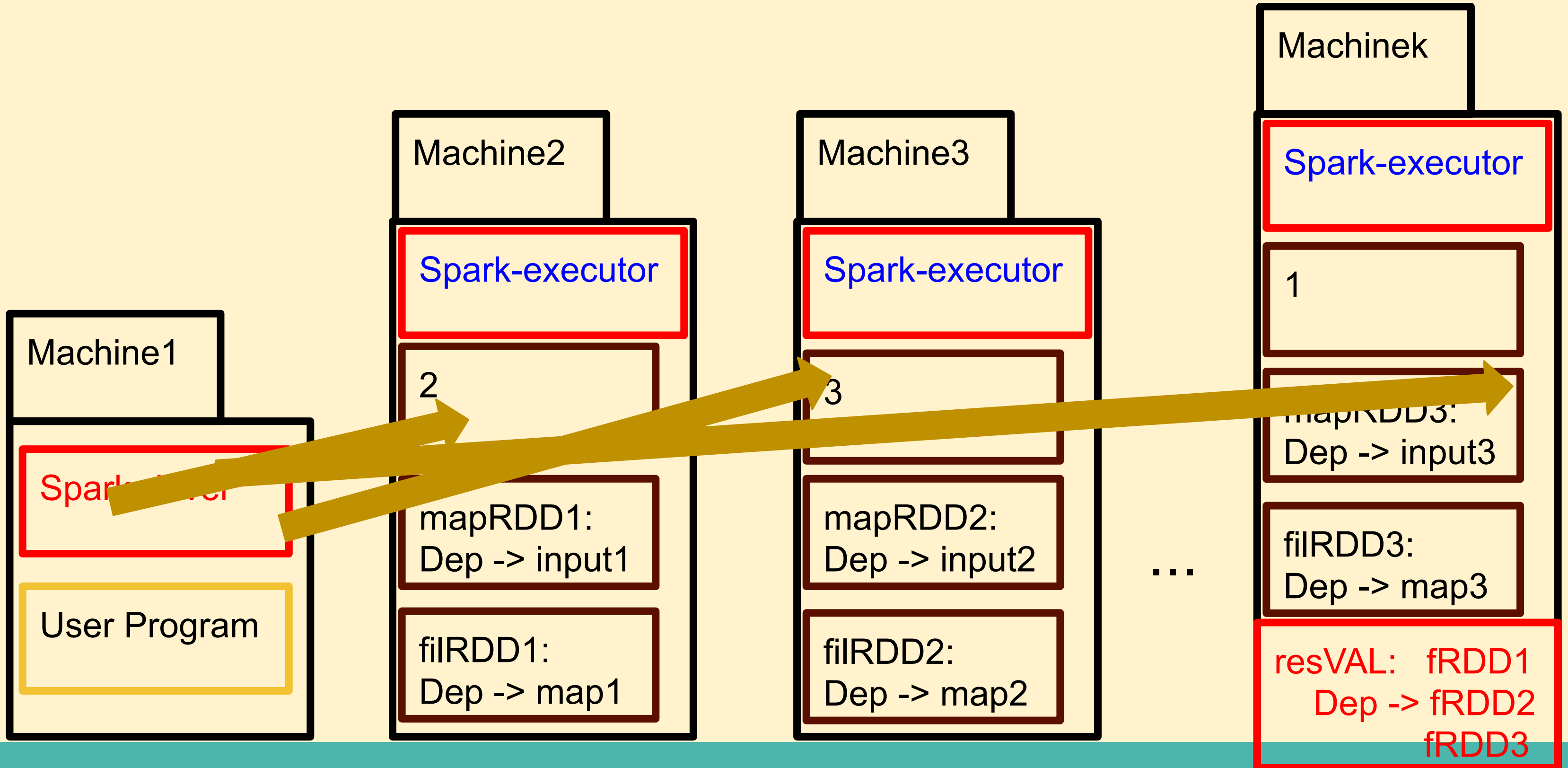
filRDD3: Dep -> map3

resVAL: fRDD1
Dep -> fRDD2
fRDD3

Lineage: Lazy Evaluation and Persistance

...this is not exactly what happens...

```
inputRDD  = sc.parallelize( [ 1, 2, 3, 4, 5 ] )
mappedRDD  = inputRDD.map( lambda elem : elem + 1 )
filteredRDD  = mappedRDD.filter( lambda elem : elem > 3 )
resVAL = filteredRDD.count()
```



Machine1

Spark-driver

User Program

Machine2

Spark-executor

2

mapRDD1:
Dep -> input1

filRDD1:
Dep -> map1

Machine3

Spark-executor

3

mapRDD2:
Dep -> input2

filRDD2:
Dep -> map2

...

Machinek

Spark-executor

1

mapRDD3:
Dep -> input3

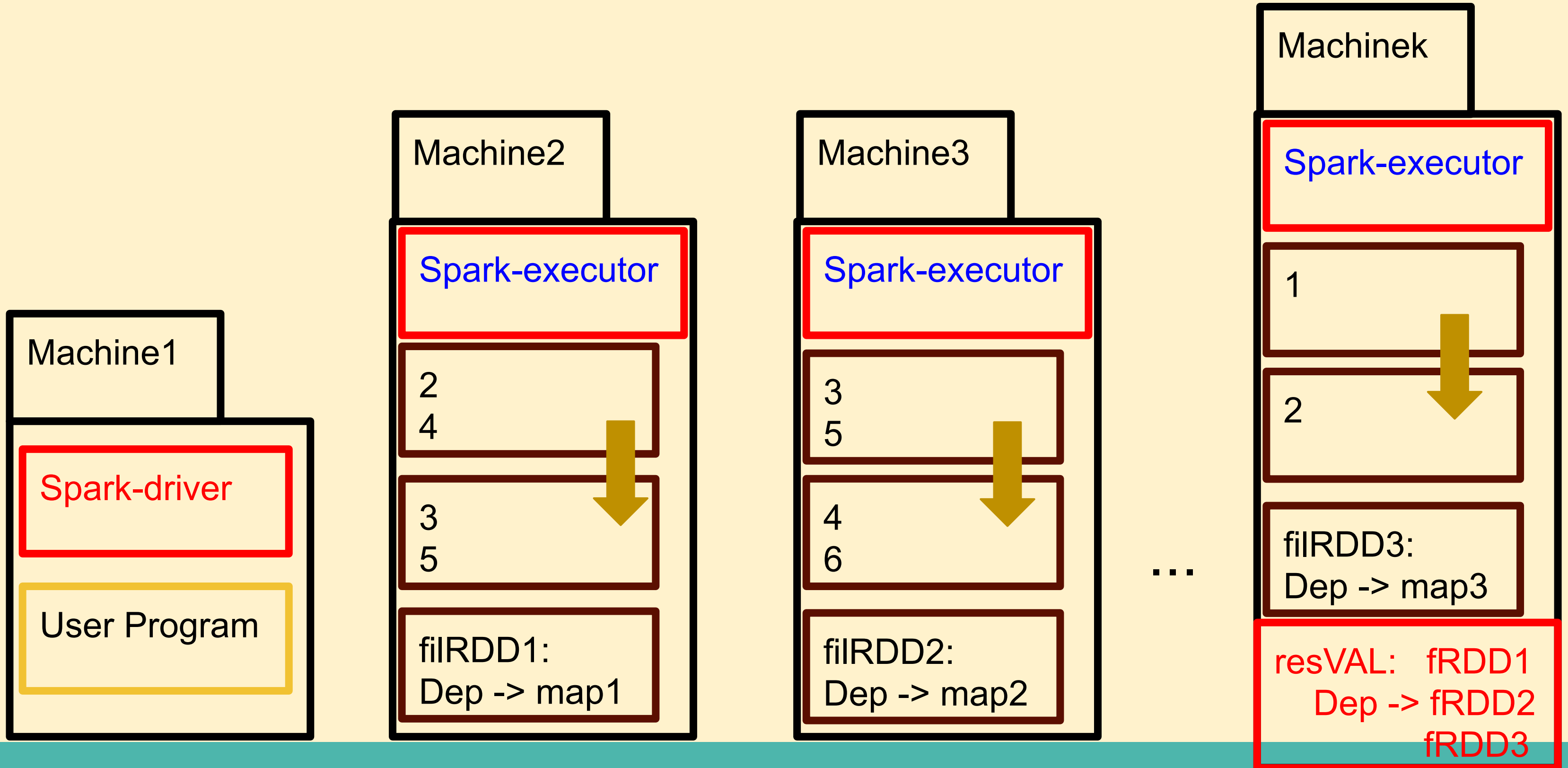
filRDD3:
Dep -> map3

resVAL: fRDD1
Dep -> fRDD2
fRDD3

Lineage: Lazy Evaluation and Persistance

...this is not exactly what happens...

```
inputRDD  = sc.parallelize( [ 1, 2, 3, 4, 5 ] )
mappedRDD  = inputRDD.map( lambda elem : elem + 1 )
filteredRDD  = mappedRDD.filter( lambda elem : elem > 3 )
resVAL = filteredRDD.count()
```



Machine1

Spark-driver

User Program

Machine2

Spark-executor

2
4

3
5

filRDD1:
Dep -> map1

Machine3

Spark-executor

3
5

4
6

filRDD2:
Dep -> map2

...

Machinek

Spark-executor

1

2

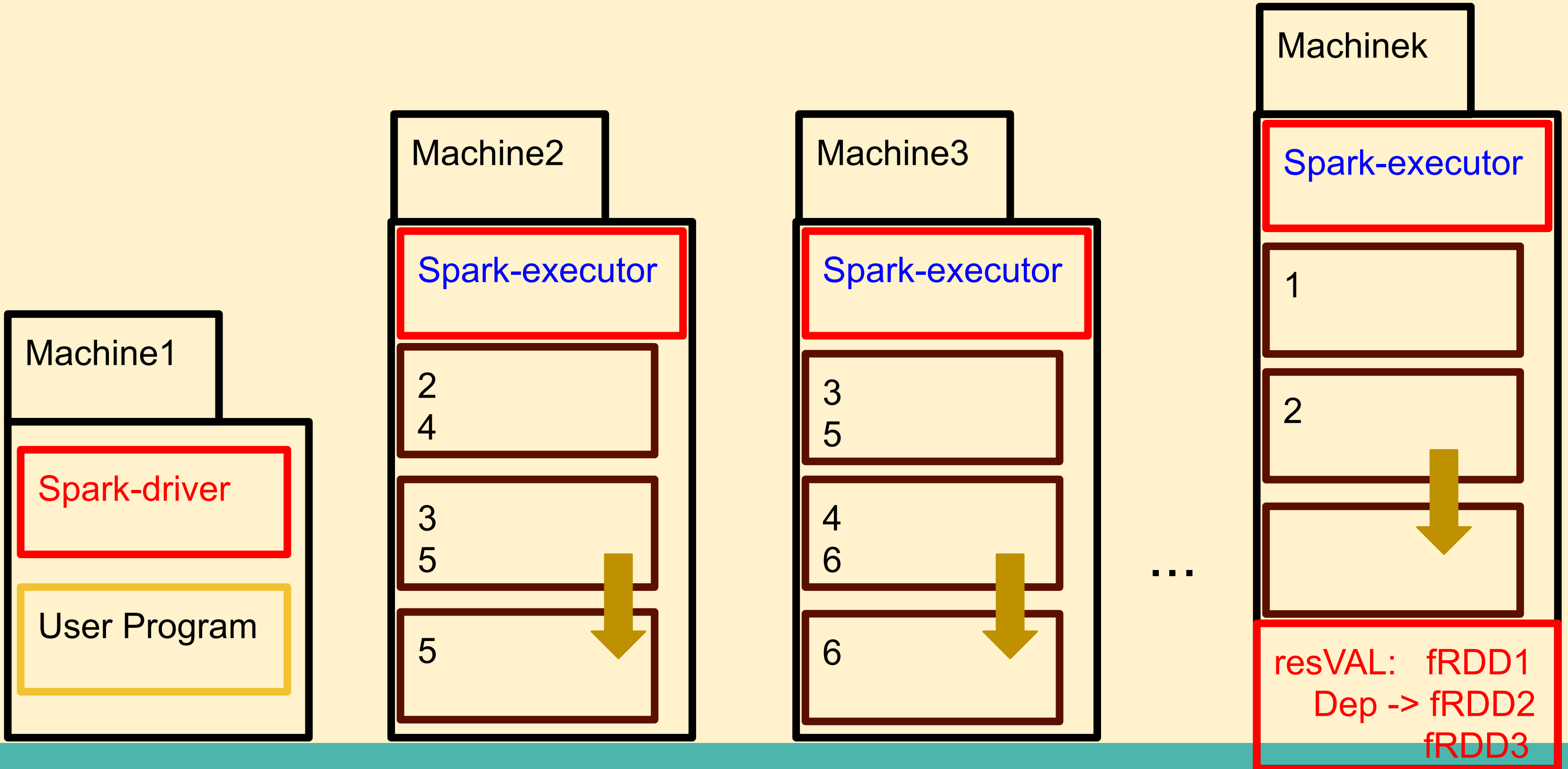
filRDD3:
Dep -> map3

resVAL: fRDD1
Dep -> fRDD2
fRDD3

Lineage: Lazy Evaluation and Persistance

…this is not exactly what happens…

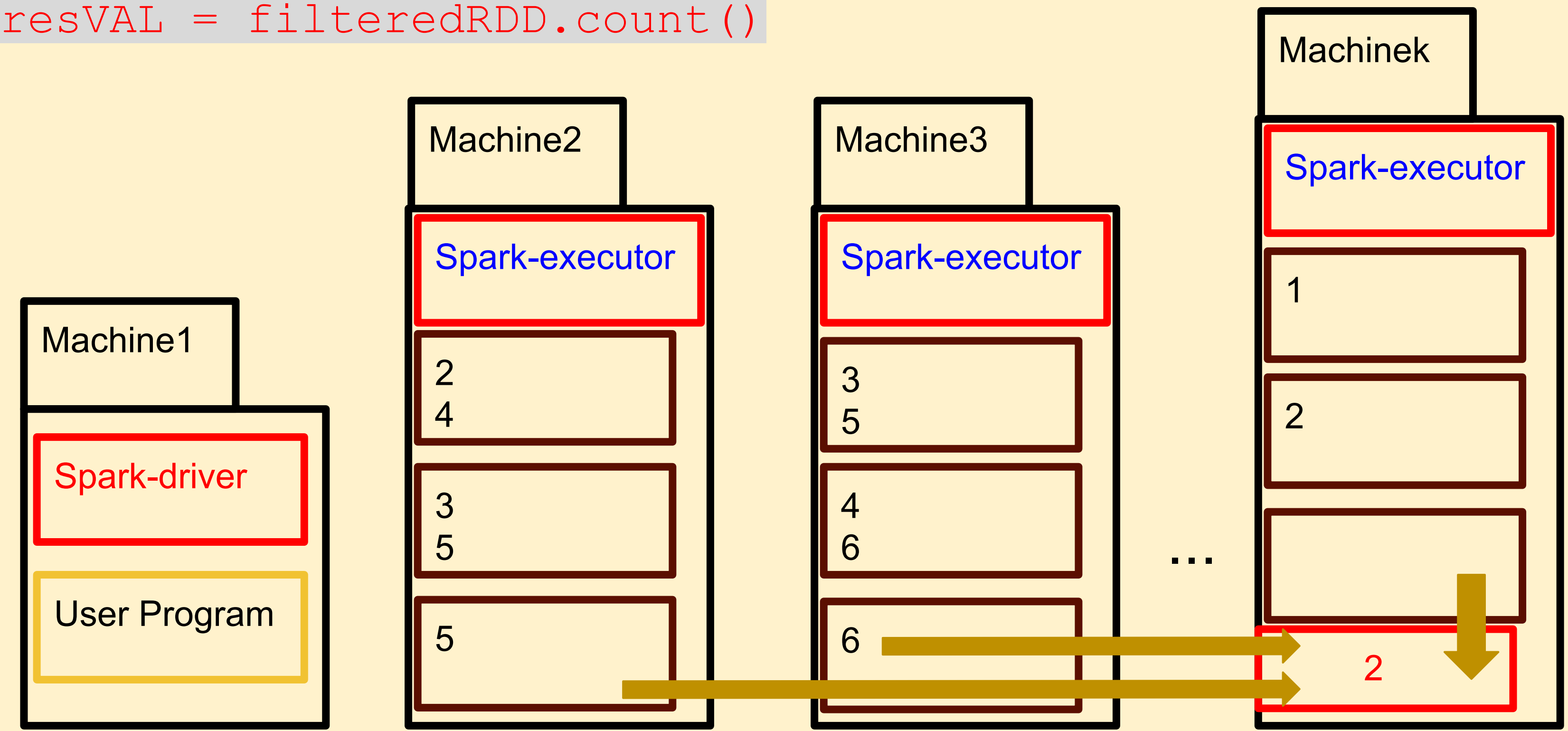
```
inputRDD  = sc.parallelize( [ 1, 2, 3, 4, 5 ] )
mappedRDD  = inputRDD.map( lambda elem : elem + 1 )
filteredRDD  = mappedRDD.filter( lambda elem : elem > 3 )
resVAL = filteredRDD.count()
```



Lineage: Lazy Evaluation

...this is not exactly what happens...

```
inputRDD   = sc.parallelize( [ 1, 2, 3, 4, 5 ] )
mappedRDD  = inputRDD.map( lambda elem : elem + 1 )
filteredRDD  = mappedRDD.filter( lambda elem : elem > 3 )
resVAL = filteredRDD.count()
```



Machine1
Spark-driver
User Program

Machine2
Spark-executor
2
4
3
5
5

Machine3
Spark-executor
3
5
4
6
6

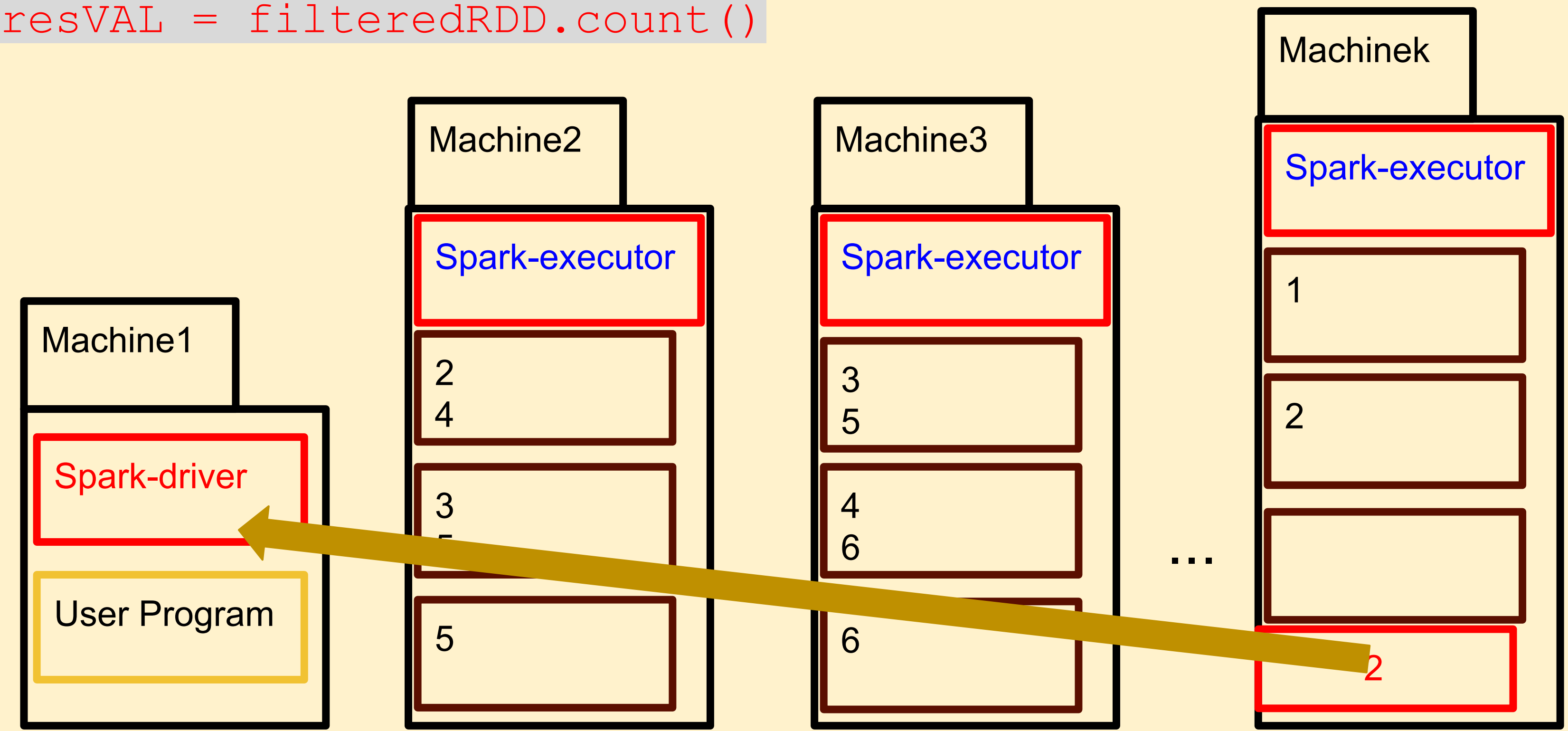
...

Machinek
Spark-executor
1
2
2

Lineage: Lazy Evaluation

…this is not exactly what happens…

```
inputRDD  = sc.parallelize( [ 1, 2, 3, 4, 5 ] )
mappedRDD  = inputRDD.map( lambda elem : elem + 1 )
filteredRDD  = mappedRDD.filter( lambda elem : elem > 3 )
resVAL = filteredRDD.count()
```



Lineage: Lazy Evaluation and Persistance

Actually, RDD partitions are not computed and kept in memory!

Instead, RDD partitions are computed (by demand), used for what they were meant to... and removed straight away afterwards!

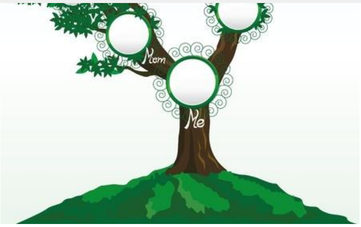
Lineage: Lazy Evaluation and Persistance

Actually, RDD partitions are not computed and kept in memory!

Instead, RDD partitions are computed (by demand), used for what they were meant to... and removed straight away afterwards!

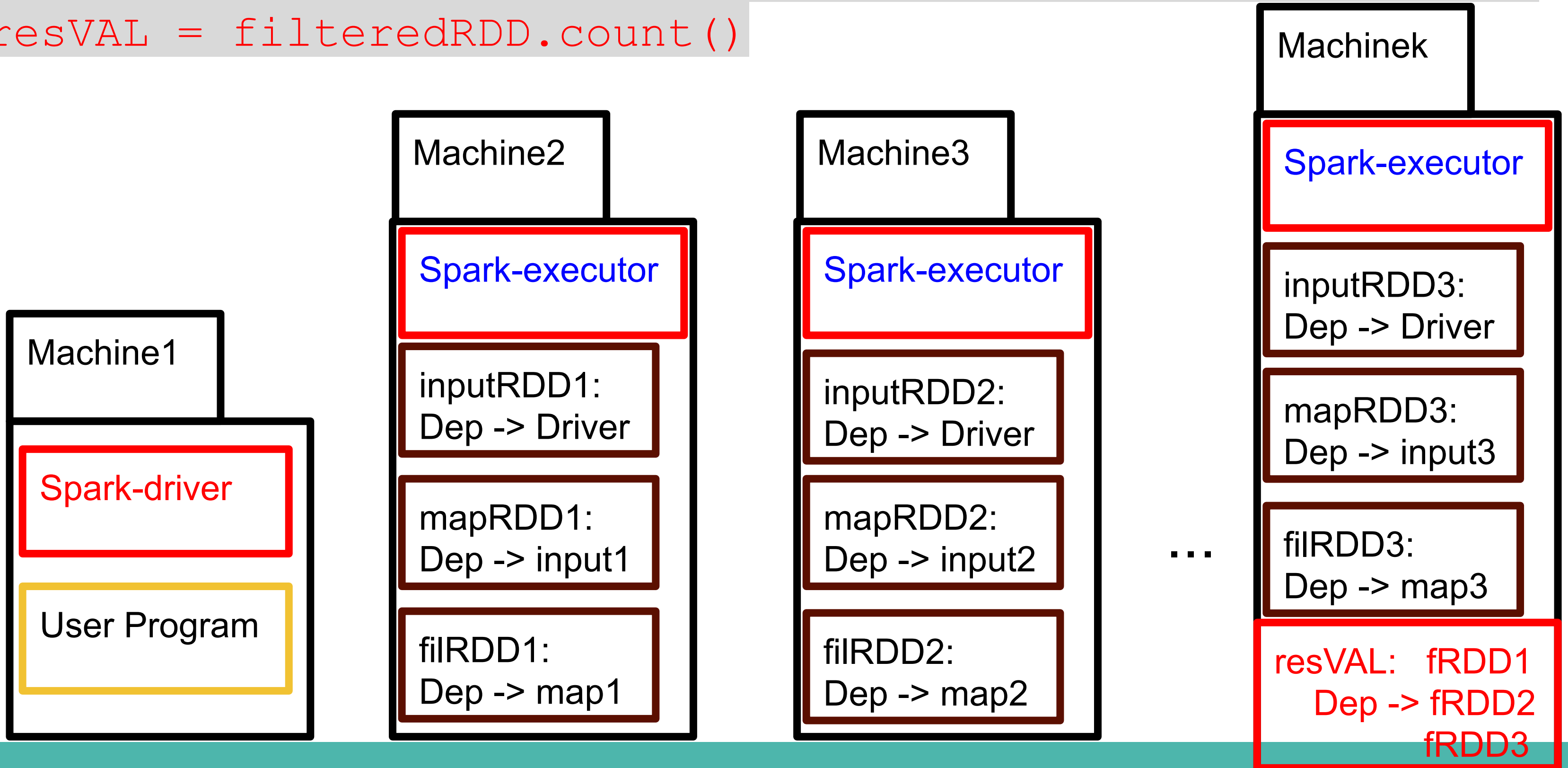
So this is what really happens!

Lineage: Lazy Evaluation and Persistance



When an **action** takes place, computation is triggered by tracing the lineage backwards.

```
inputRDD   = sc.parallelize( [ 1, 2, 3, 4, 5 ] )
mappedRDD  = inputRDD.map( lambda elem : elem + 1 )
filteredRDD  = mappedRDD.filter( lambda elem : elem > 3 )
resVAL = filteredRDD.count()
```



Machine1

Spark-driver

User Program

Machine2

Spark-executor

inputRDD1: Dep -> Driver

mapRDD1: Dep -> input1

filRDD1: Dep -> map1

Machine3

Spark-executor

inputRDD2: Dep -> Driver

mapRDD2: Dep -> input2

filRDD2: Dep -> map2

…

Machinek

Spark-executor

inputRDD3: Dep -> Driver

mapRDD3: Dep -> input3

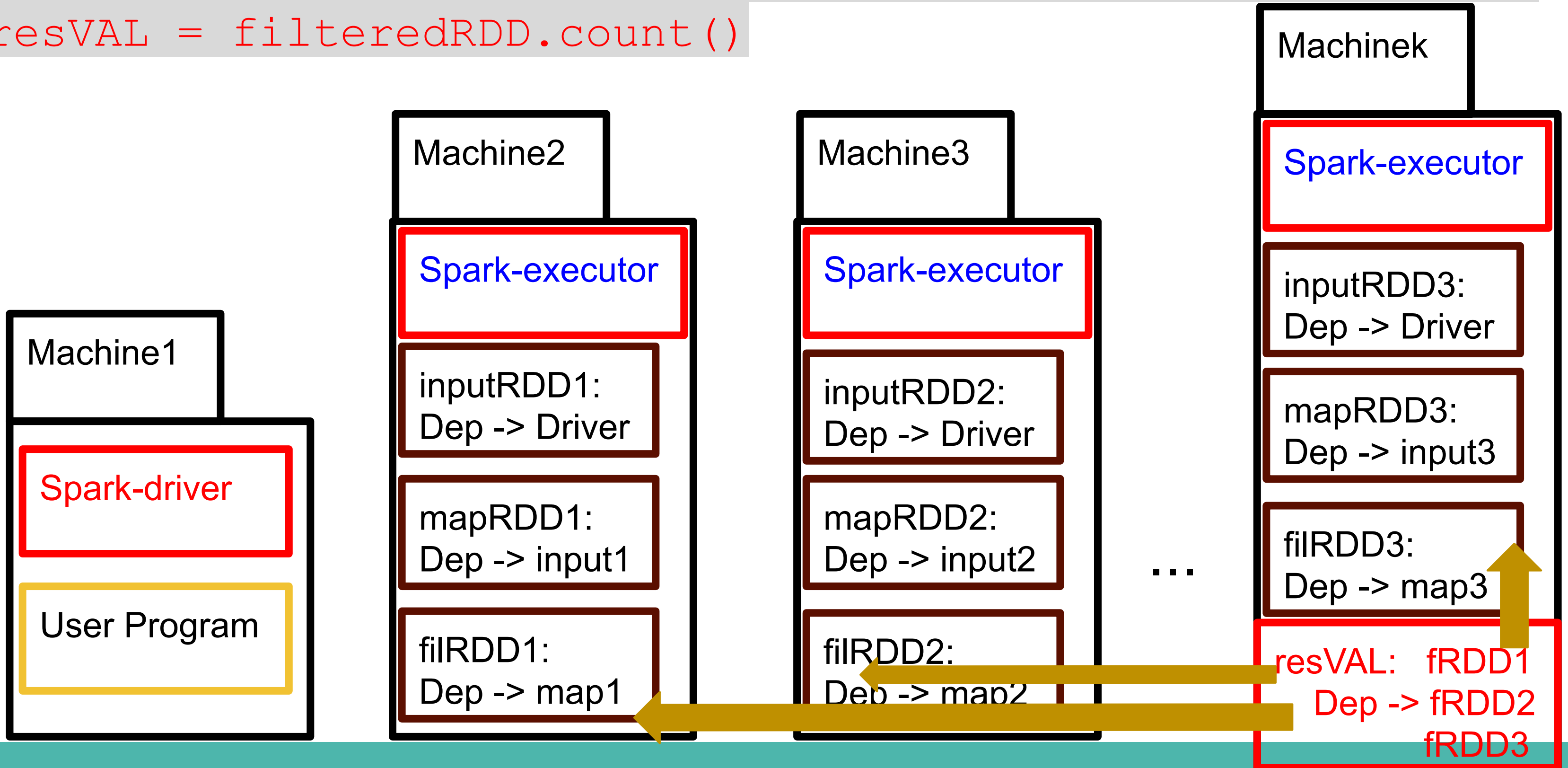
filRDD3: Dep -> map3

resVAL: fRDD1
Dep -> fRDD2
fRDD3

Lineage: Lazy Evaluation and Persistance

Who do I depend on?

```
inputRDD   = sc.parallelize( [ 1, 2, 3, 4, 5 ] )
mappedRDD  = inputRDD.map( lambda elem : elem + 1 )
filteredRDD  = mappedRDD.filter( lambda elem : elem > 3 )
resVAL = filteredRDD.count()
```



Machine1

Spark-driver

User Program

Machine2

Spark-executor

inputRDD1:
Dep -> Driver

mapRDD1:
Dep -> input1

filRDD1:
Dep -> map1

Machine3

Spark-executor

inputRDD2:
Dep -> Driver

mapRDD2:
Dep -> input2

filRDD2:
Dep -> map2

...

Machinek

Spark-executor

inputRDD3:
Dep -> Driver

mapRDD3:
Dep -> input3

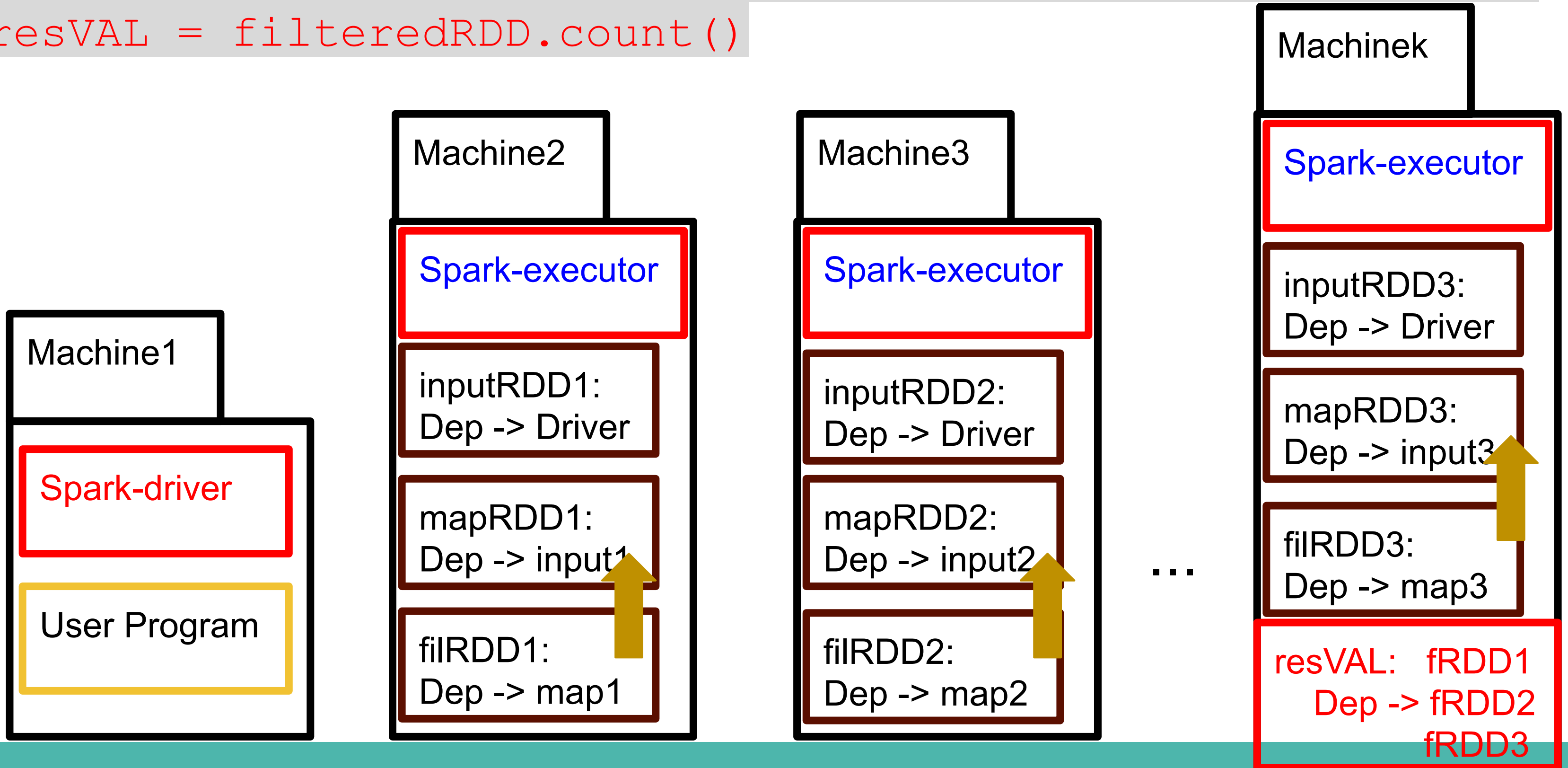
filRDD3:
Dep -> map3

resVAL: fRDD1
Dep -> fRDD2
fRDD3

Lineage: Lazy Evaluation and Persistance

And, likewise, who do these RDD partitions depend on?

```
inputRDD  = sc.parallelize( [ 1, 2, 3, 4, 5 ] )
mappedRDD  = inputRDD.map( lambda elem : elem + 1 )
filteredRDD  = mappedRDD.filter( lambda elem : elem > 3 )
resVAL = filteredRDD.count()
```



Machine1

Spark-driver

User Program

Machine2

Spark-executor

inputRDD1:
Dep -> Driver

mapRDD1:
Dep -> input1

filRDD1:
Dep -> map1

Machine3

Spark-executor

inputRDD2:
Dep -> Driver

mapRDD2:
Dep -> input2

filRDD2:
Dep -> map2

...

Machinek

Spark-executor

inputRDD3:
Dep -> Driver

mapRDD3:
Dep -> input3

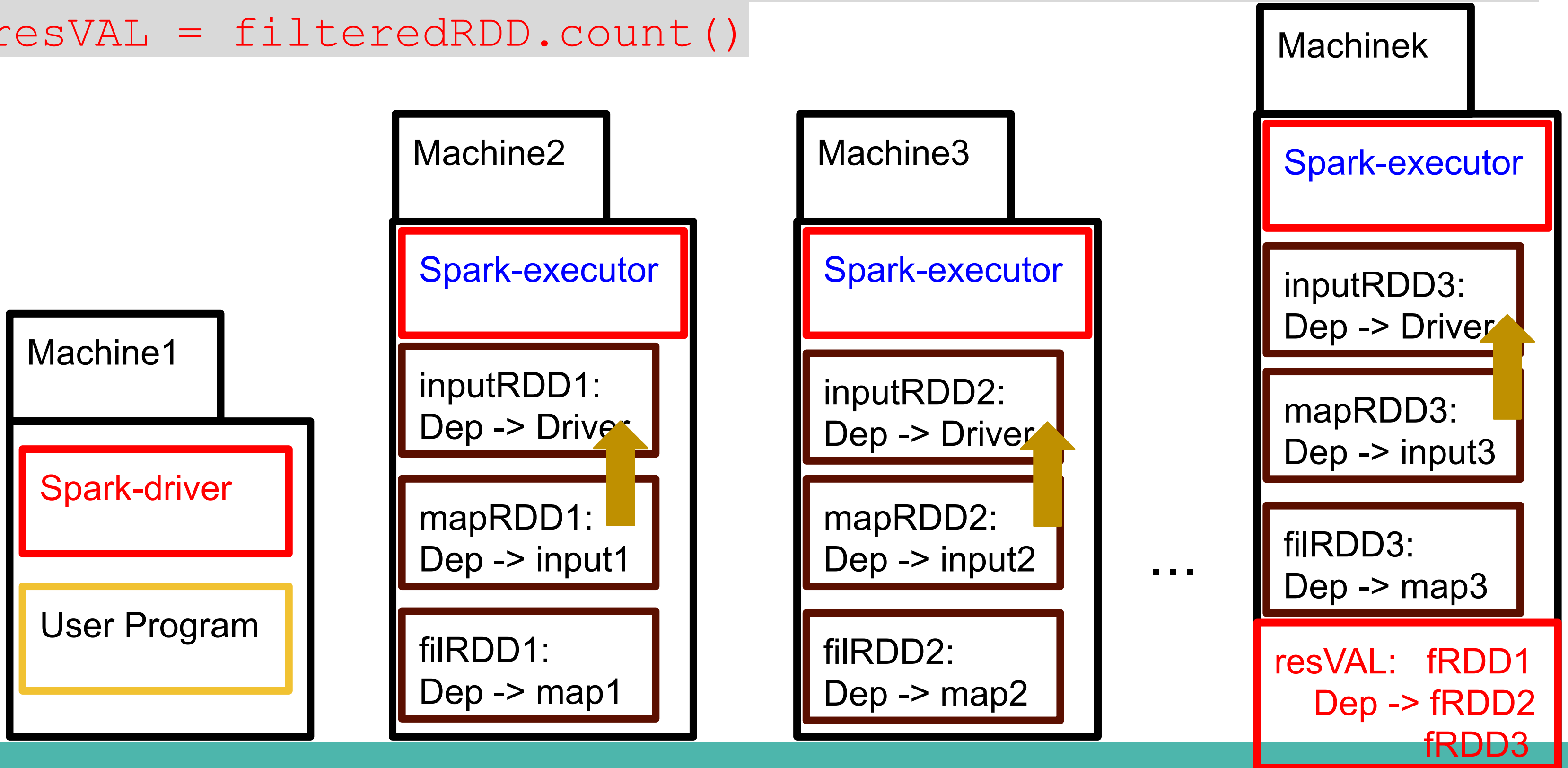
filRDD3:
Dep -> map3

resVAL: fRDD1
Dep -> fRDD2
fRDD3

Lineage: Lazy Evaluation and Persistance

And so on and so on…

```
inputRDD   = sc.parallelize( [ 1, 2, 3, 4, 5 ] )
mappedRDD   = inputRDD.map( lambda elem : elem + 1 )
filteredRDD   = mappedRDD.filter( lambda elem : elem > 3 )
resVAL = filteredRDD.count()
```



Machine1

Spark-driver

User Program

Machine2

Spark-executor

inputRDD1: Dep -> Driver

mapRDD1: Dep -> input1

filRDD1: Dep -> map1

Machine3

Spark-executor

inputRDD2: Dep -> Driver

mapRDD2: Dep -> input2

filRDD2: Dep -> map2

…

Machinek

Spark-executor

inputRDD3: Dep -> Driver

mapRDD3: Dep -> input3

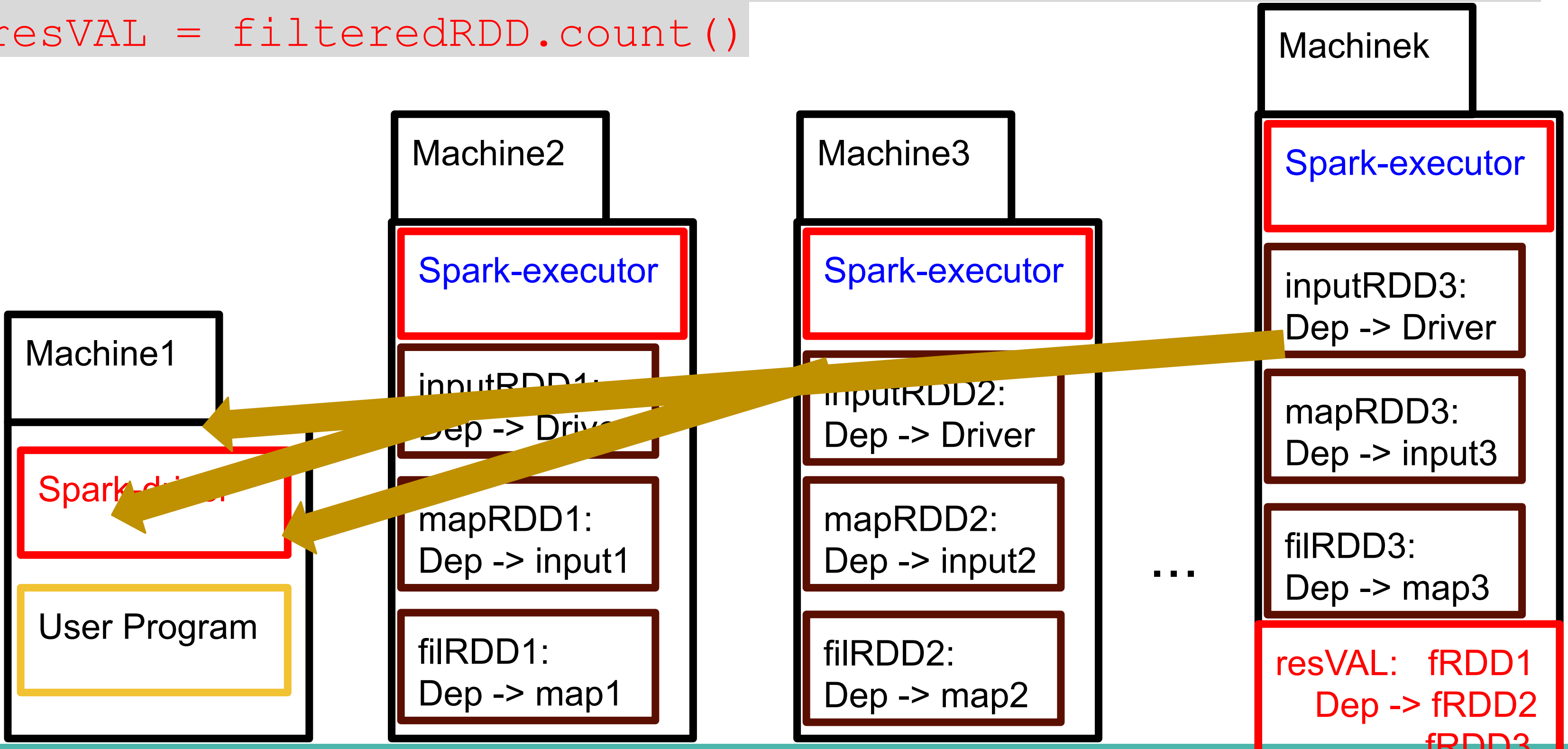
filRDD3: Dep -> map3

resVAL: fRDD1
Dep -> fRDD2
fRDD3

Lineage: Lazy Evaluation and Persistance

And so on and so on…

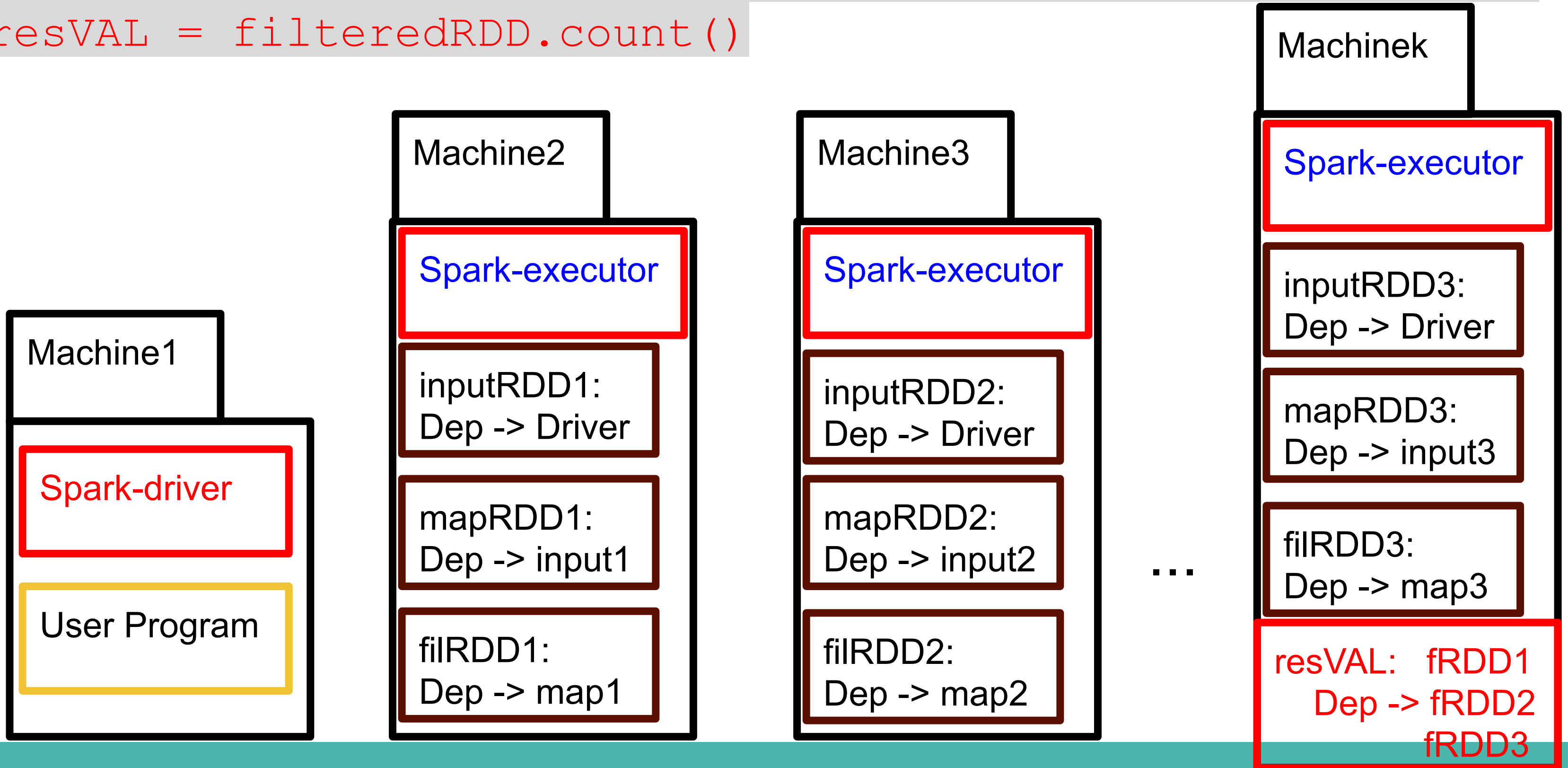
```
inputRDD   = sc.parallelize( [ 1, 2, 3, 4, 5 ] )
mappedRDD  = inputRDD.map( lambda elem : elem + 1 )
filteredRDD  = mappedRDD.filter( lambda elem : elem > 3 )
resVAL = filteredRDD.count()
```



Lineage: Lazy Evaluation and Persistance

And now that I know the full lineage, computation can start lazily.

```
inputRDD   = sc.parallelize( [ 1, 2, 3, 4, 5 ] )
mappedRDD  = inputRDD.map( lambda elem : elem + 1 )
filteredRDD  = mappedRDD.filter( lambda elem : elem > 3 )
resVAL = filteredRDD.count()
```



Machine1

Spark-driver

User Program

Machine2

Spark-executor

inputRDD1: Dep -> Driver

mapRDD1: Dep -> input1

filRDD1: Dep -> map1

Machine3

Spark-executor

inputRDD2: Dep -> Driver

mapRDD2: Dep -> input2

filRDD2: Dep -> map2

…

Machinek

Spark-executor

inputRDD3: Dep -> Driver

mapRDD3: Dep -> input3

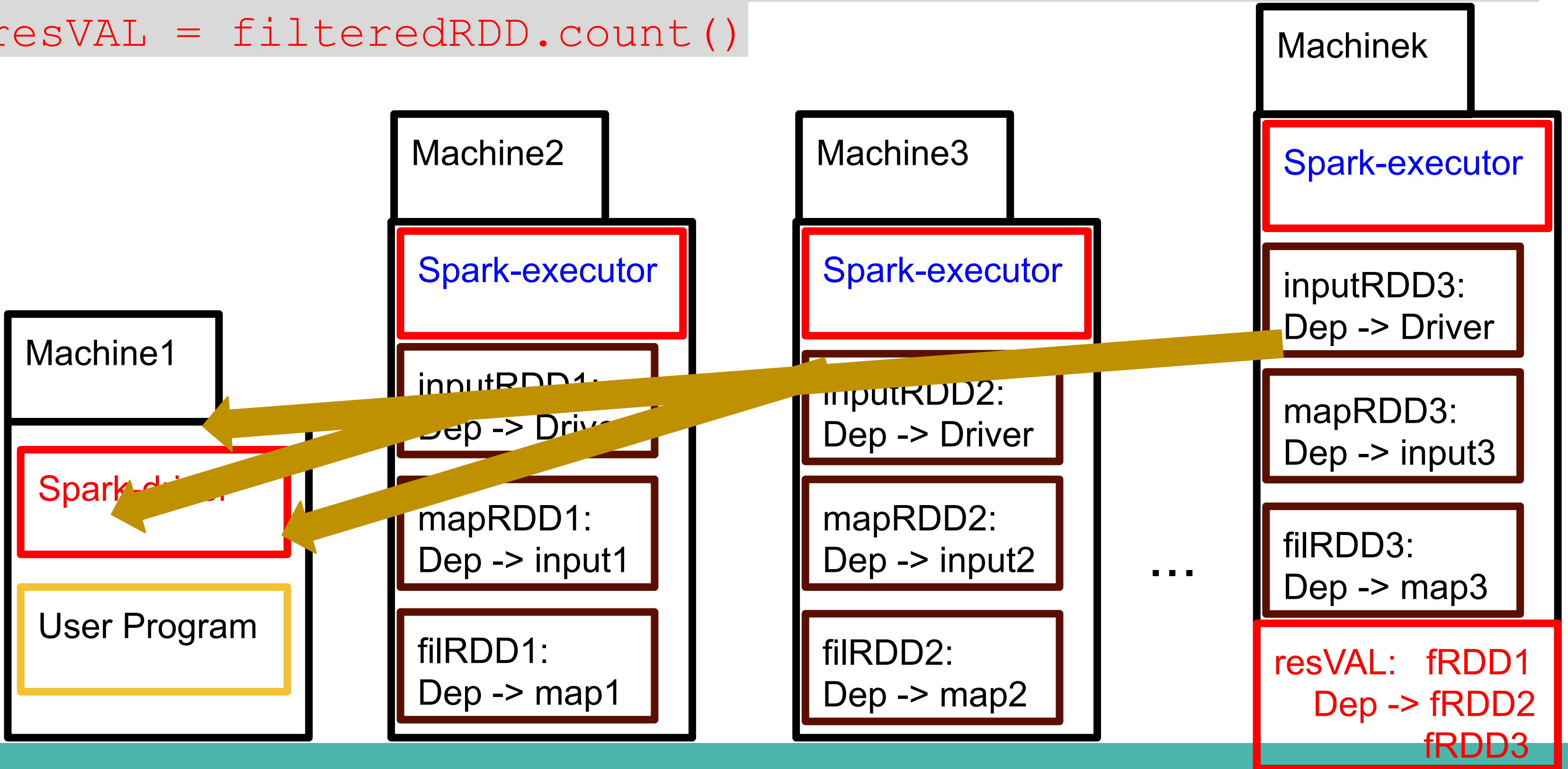
filRDD3: Dep -> map3

resVAL: fRDD1 Dep -> fRDD2 fRDD3

Lineage: Lazy Evaluation and Persistance

Ok, inputRDD is needed, so it is computed using the driver.

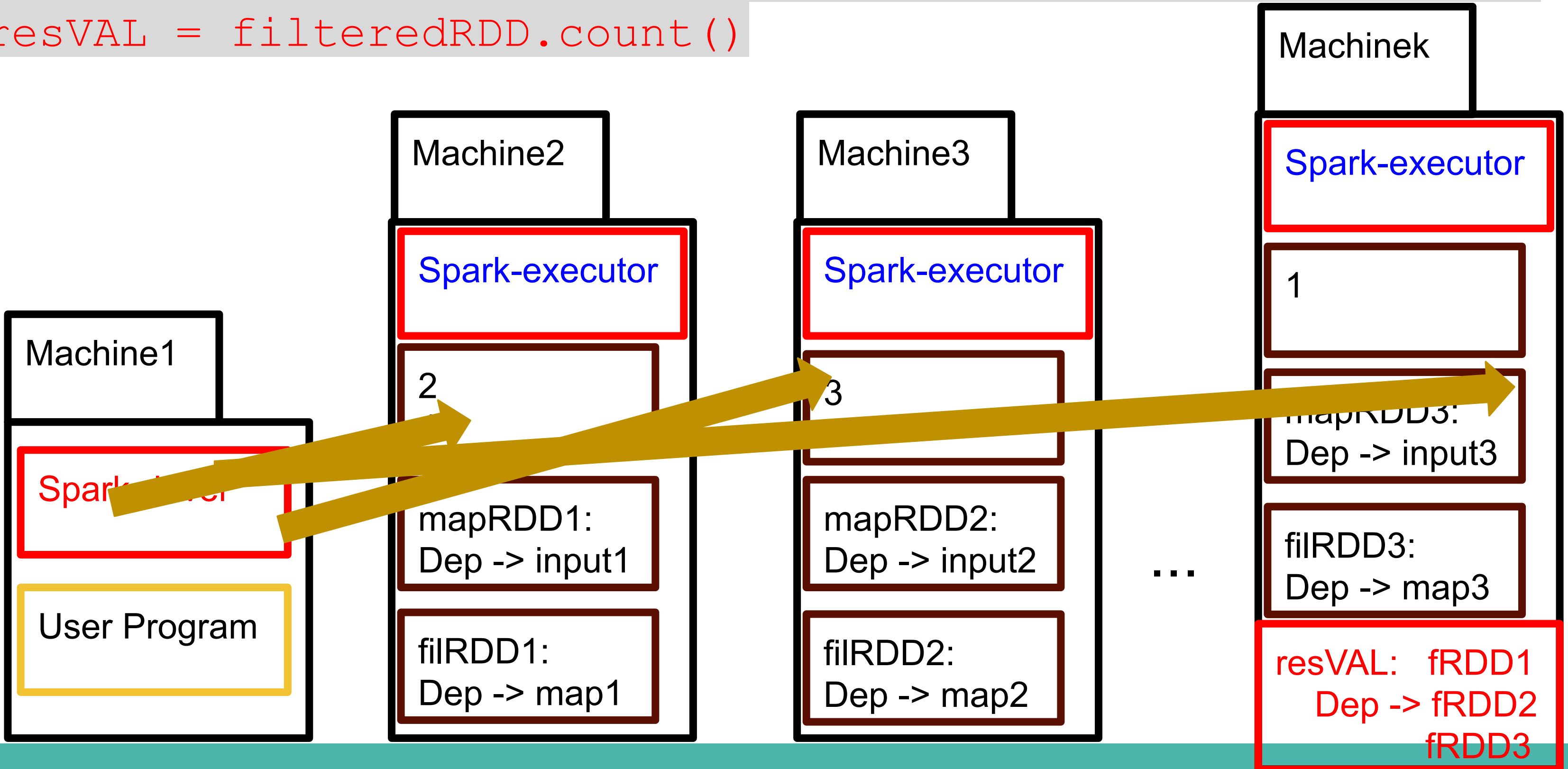
```
inputRDD   = sc.parallelize( [ 1, 2, 3, 4, 5 ] )
mappedRDD  = inputRDD.map( lambda elem : elem + 1 )
filteredRDD  = mappedRDD.filter( lambda elem : elem > 3 )
resVAL = filteredRDD.count()
```



Lineage: Lazy Evaluation and Persistance

Ok, inputRDD is needed, so it is computed using the driver.

```
inputRDD  = sc.parallelize( [ 1, 2, 3, 4, 5 ] )
mappedRDD  = inputRDD.map( lambda elem : elem + 1 )
filteredRDD  = mappedRDD.filter( lambda elem : elem > 3 )
resVAL = filteredRDD.count()
```



Machine1

Spark-driver

User Program

Machine2

Spark-executor

2

mapRDD1:
Dep -> input1

filRDD1:
Dep -> map1

Machine3

Spark-executor

3

mapRDD2:
Dep -> input2

filRDD2:
Dep -> map2

…

Machinek

Spark-executor

1

mapRDD3:
Dep -> input3

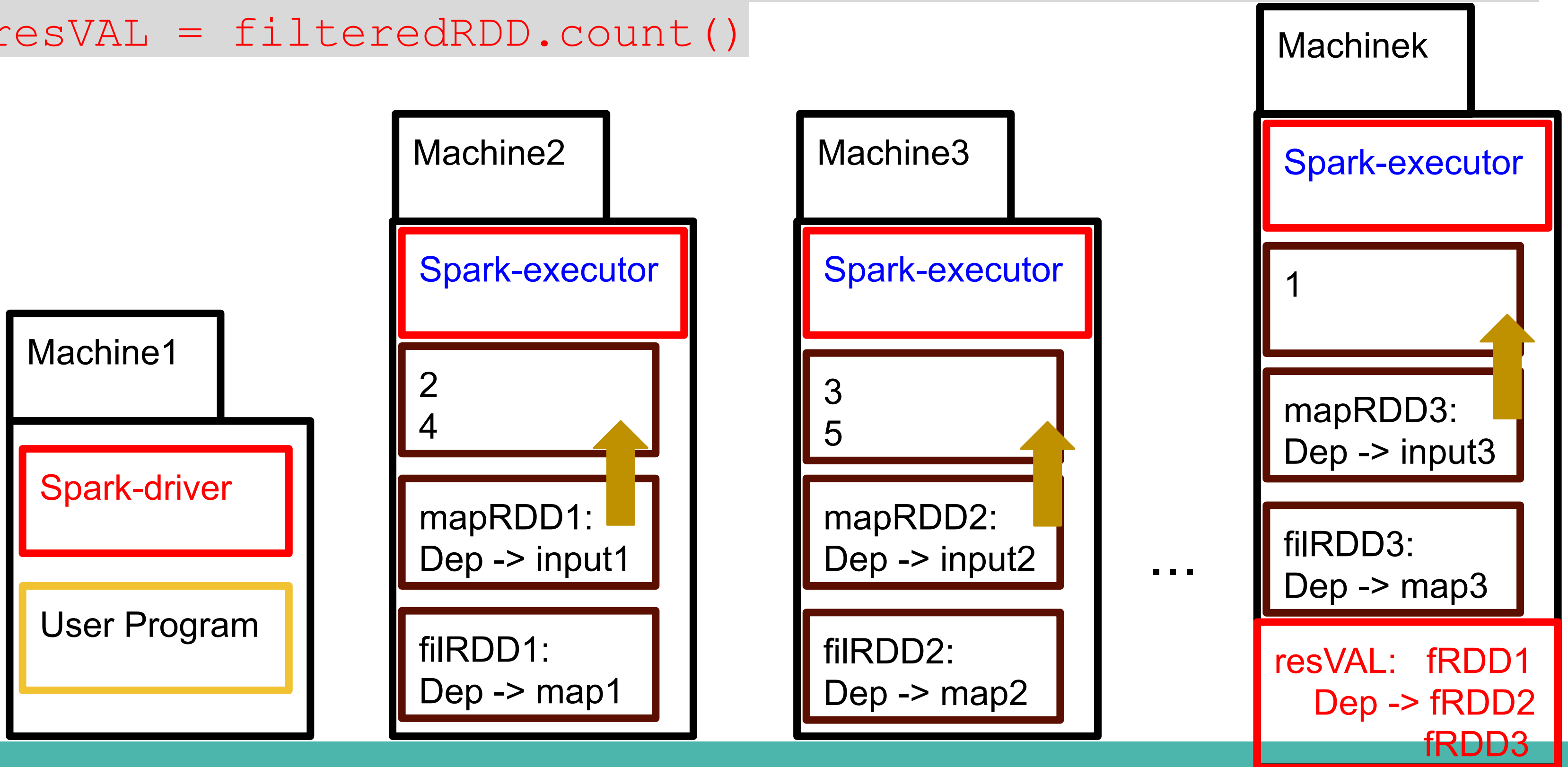
filRDD3:
Dep -> map3

resVAL: fRDD1
Dep -> fRDD2
fRDD3

Lineage: Lazy Evaluation and Persistance

Ok, mapRDD is needed, so it is computed using inputRDD.

```
inputRDD  = sc.parallelize( [ 1, 2, 3, 4, 5 ] )
mappedRDD  = inputRDD.map( lambda elem : elem + 1 )
filteredRDD  = mappedRDD.filter( lambda elem : elem > 3 )
resVAL = filteredRDD.count()
```



Machine1

Spark-driver

User Program

Machine2

Spark-executor

2
4

mapRDD1:
Dep -> input1

filRDD1:
Dep -> map1

Machine3

Spark-executor

3
5

mapRDD2:
Dep -> input2

filRDD2:
Dep -> map2

…

Machinek

Spark-executor

1

mapRDD3:
Dep -> input3

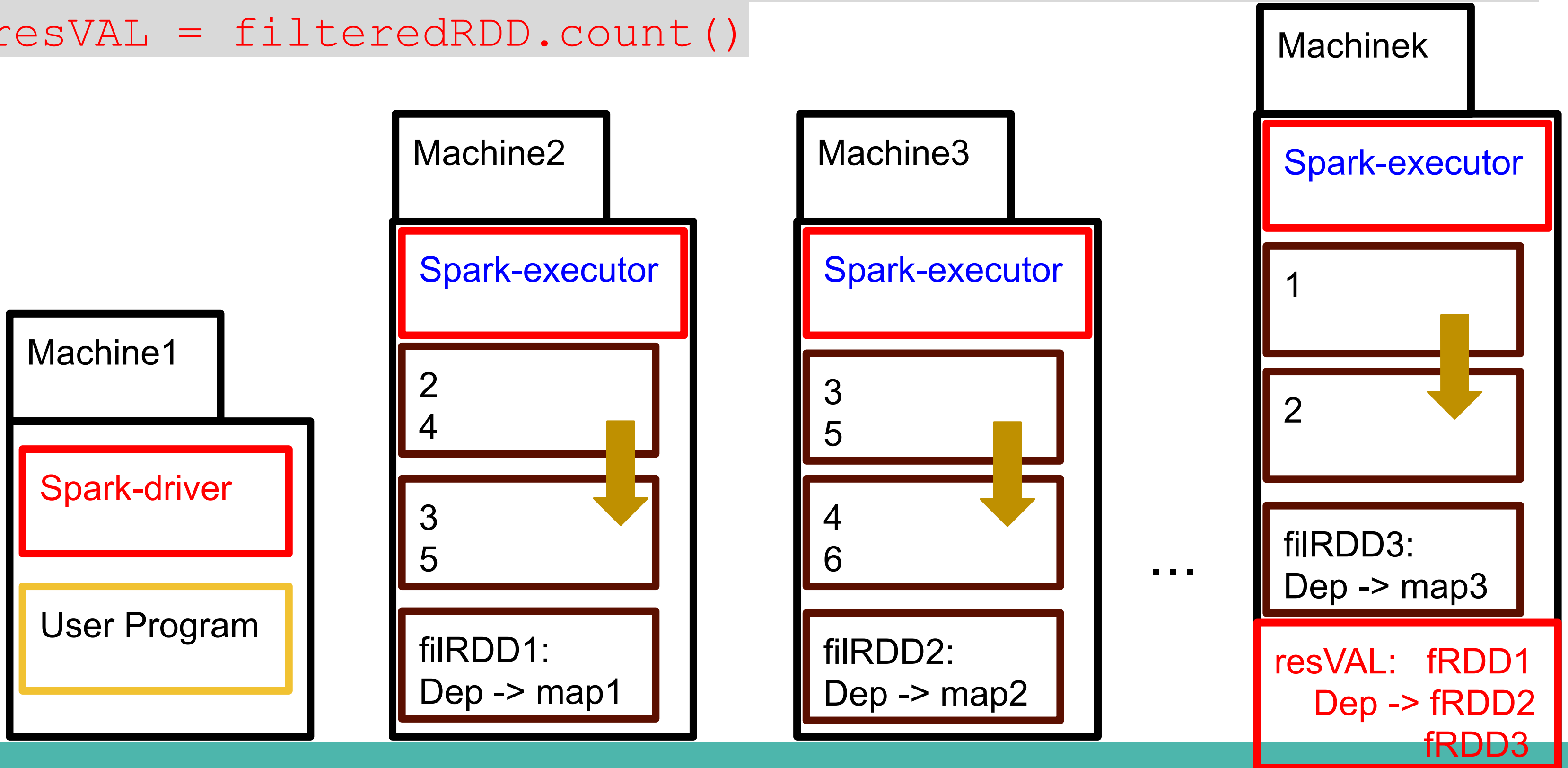
filRDD3:
Dep -> map3

resVAL: fRDD1
Dep -> fRDD2
fRDD3

Lineage: Lazy Evaluation and Persistance

Ok, mapRDD is needed, so it is computed using inputRDD.

```
inputRDD  = sc.parallelize( [ 1, 2, 3, 4, 5 ] )
mappedRDD  = inputRDD.map( lambda elem : elem + 1 )
filteredRDD  = mappedRDD.filter( lambda elem : elem > 3 )
resVAL = filteredRDD.count()
```



Machine1

Spark-driver

User Program

Machine2

Spark-executor

2
4

3
5

filRDD1:
Dep -> map1

Machine3

Spark-executor

3
5

4
6

filRDD2:
Dep -> map2

…

Machinek

Spark-executor

1

2

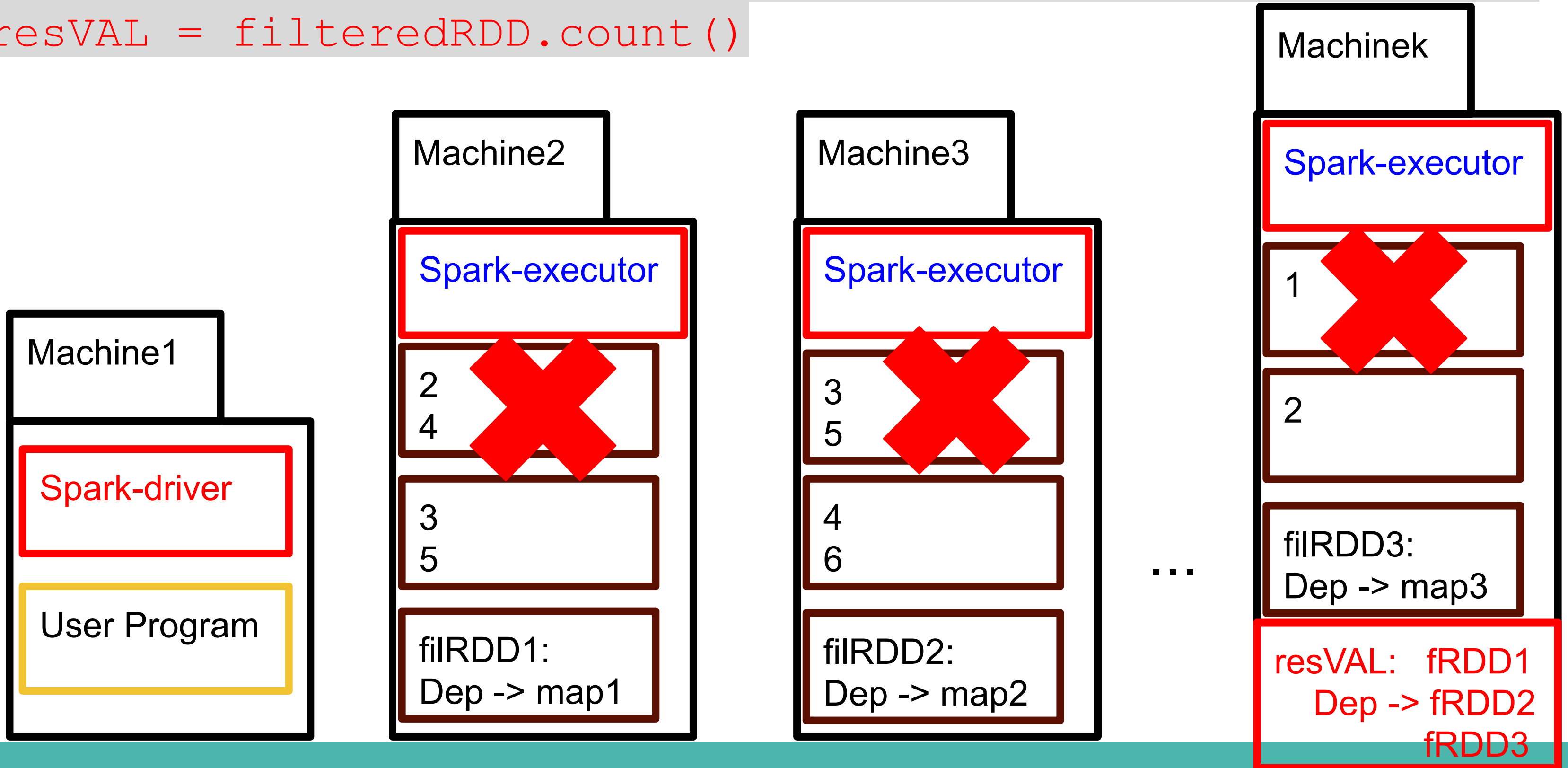
filRDD3:
Dep -> map3

resVAL: fRDD1
Dep -> fRDD2
fRDD3

Lineage: Lazy Evaluation and Persistance

Once inputRDD has been used, it is removed straight away!

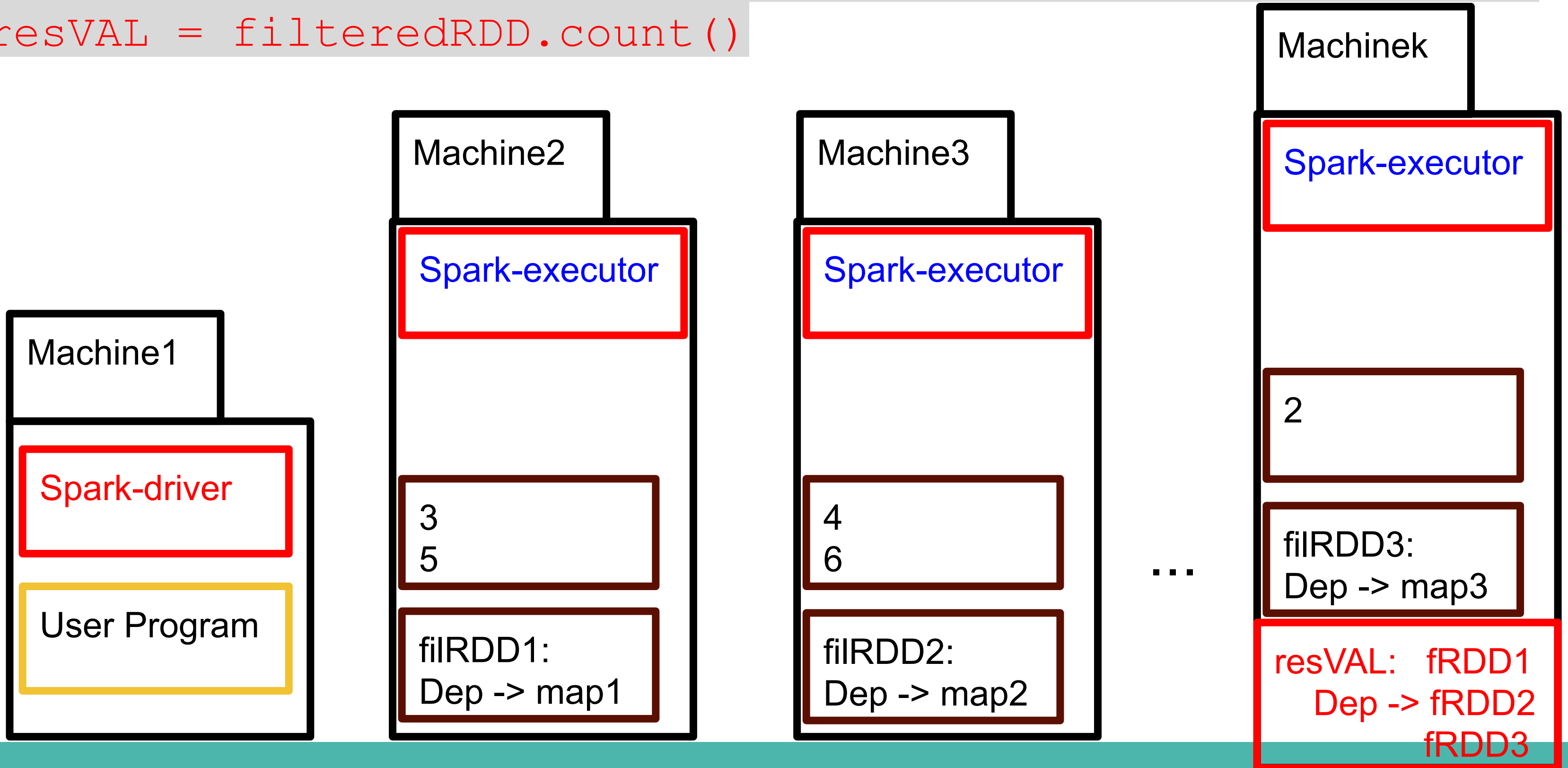
```
inputRDD  = sc.parallelize( [ 1, 2, 3, 4, 5 ] )
mappedRDD  = inputRDD.map( lambda elem : elem + 1 )
filteredRDD  = mappedRDD.filter( lambda elem : elem > 3 )
resVAL = filteredRDD.count()
```



Lineage: Lazy Evaluation and Persistance

Once inputRDD has been used, it is removed straight away!

```
inputRDD   = sc.parallelize( [ 1, 2, 3, 4, 5 ] )
mappedRDD   = inputRDD.map( lambda elem : elem + 1 )
filteredRDD   = mappedRDD.filter( lambda elem : elem > 3 )
resVAL = filteredRDD.count()
```



Machine1

Spark-driver

User Program

Machine2

Spark-executor

3
5

filRDD1:
Dep -> map1

Machine3

Spark-executor

4
6

filRDD2:
Dep -> map2

…

Machinek

Spark-executor

2

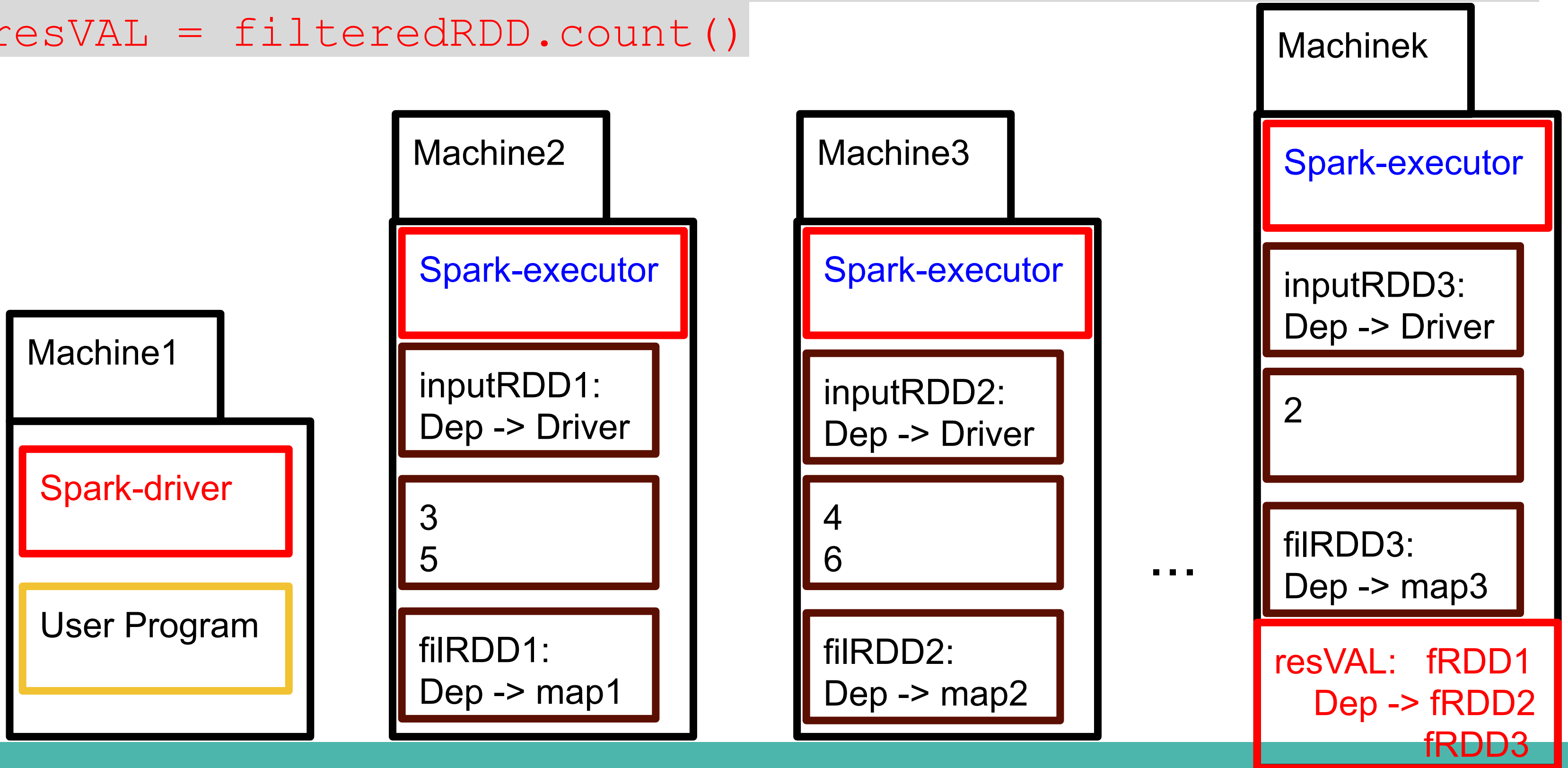
filRDD3:
Dep -> map3

resVAL: fRDD1
Dep -> fRDD2
fRDD3

Lineage: Lazy Evaluation and Persistance

Indeed, the data of inputRDD is removed, but its lineage metadata is kept.

```
inputRDD  = sc.parallelize( [ 1, 2, 3, 4, 5 ] )
mappedRDD  = inputRDD.map( lambda elem : elem + 1 )
filteredRDD  = mappedRDD.filter( lambda elem : elem > 3 )
resVAL = filteredRDD.count()
```



Machine1

Spark-driver

User Program

Machine2

Spark-executor

inputRDD1:
Dep -> Driver

3
5

filRDD1:
Dep -> map1

Machine3

Spark-executor

inputRDD2:
Dep -> Driver

4
6

filRDD2:
Dep -> map2

…

Machinek

Spark-executor

inputRDD3:
Dep -> Driver

2

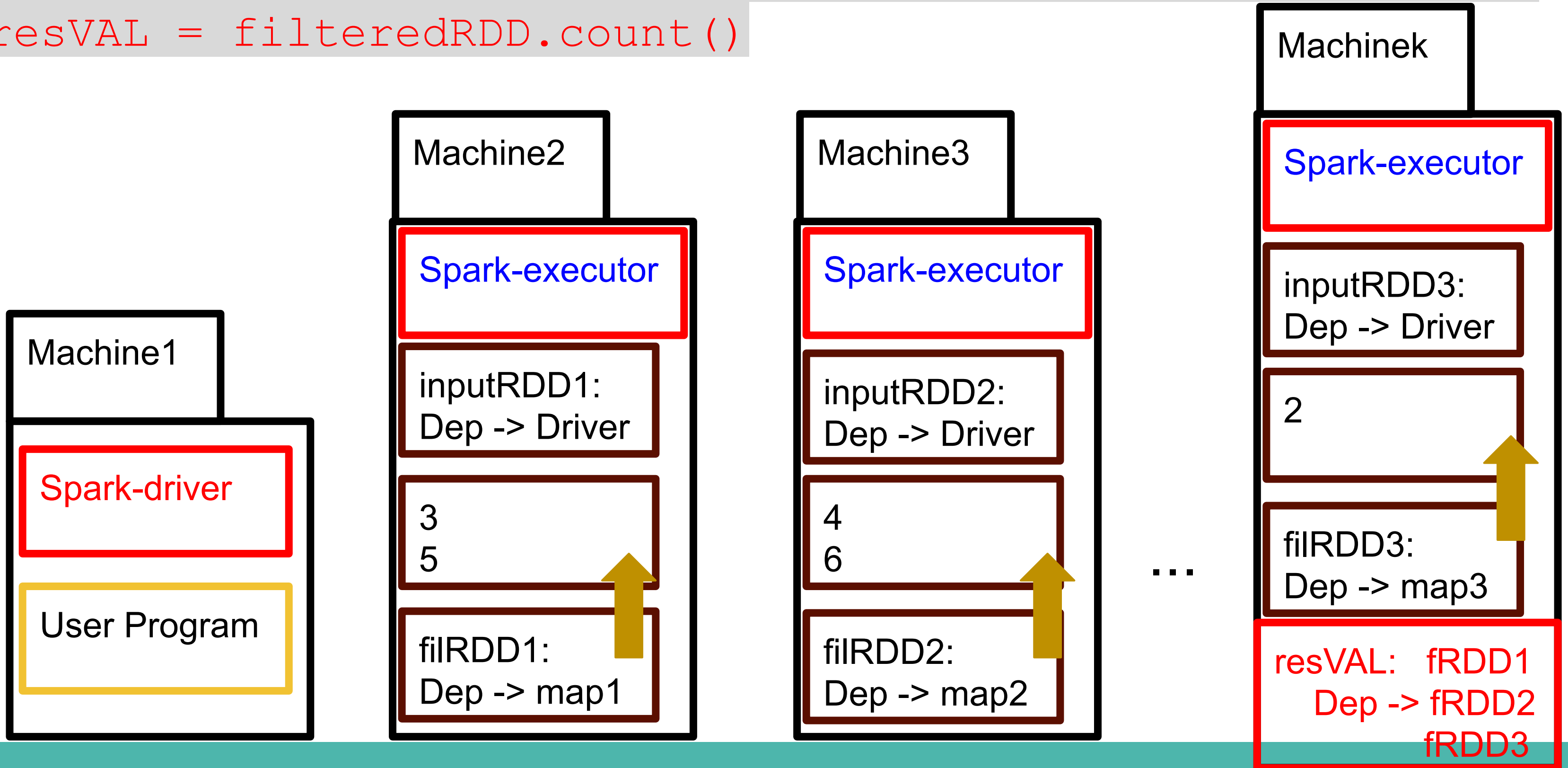
filRDD3:
Dep -> map3

resVAL: fRDD1
Dep -> fRDD2
fRDD3

Lineage: Lazy Evaluation and Persistance

Ok, filterRDD is needed, so it is computed using mapRDD.

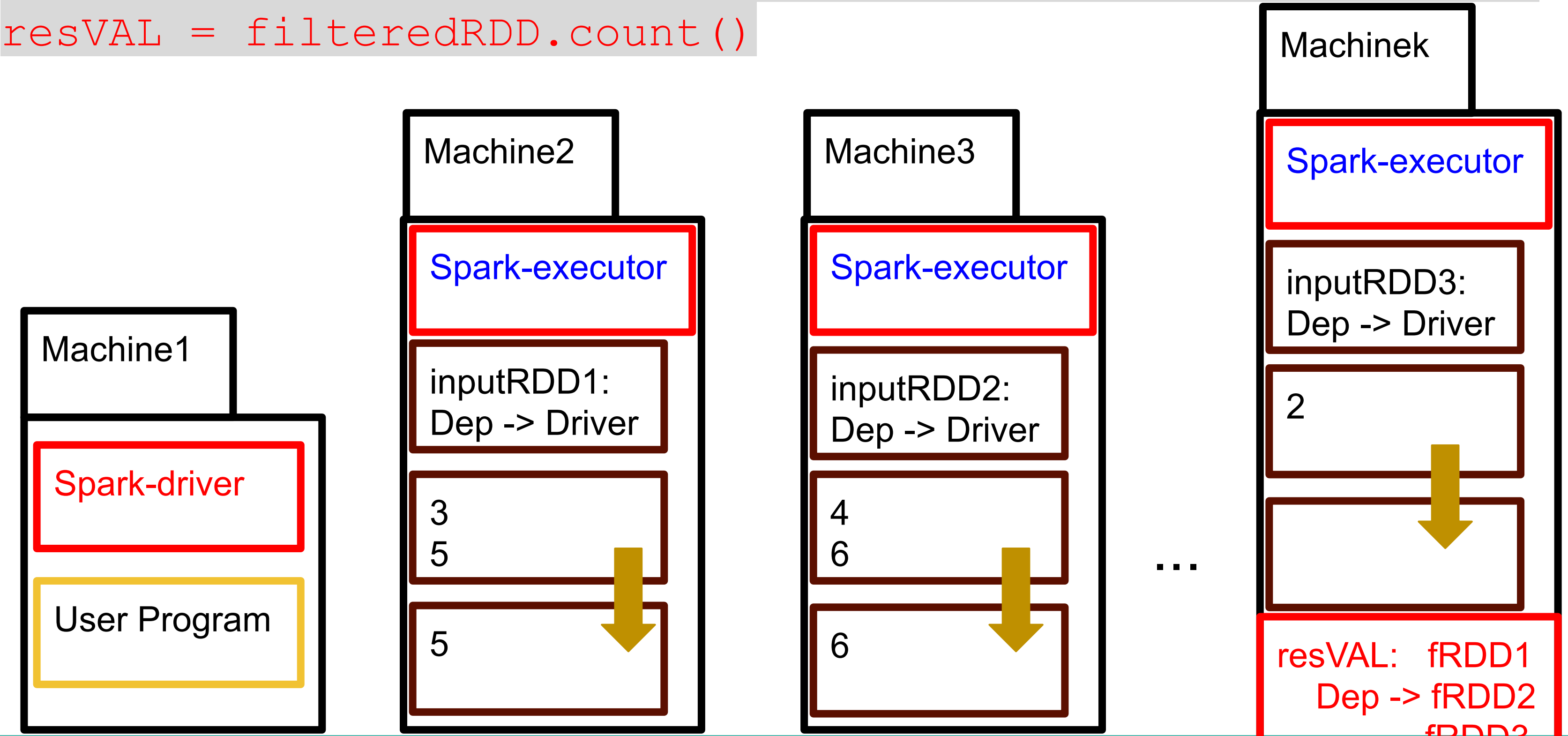
```
inputRDD   = sc.parallelize( [ 1, 2, 3, 4, 5 ] )
mappedRDD  = inputRDD.map( lambda elem : elem + 1 )
filteredRDD  = mappedRDD.filter( lambda elem : elem > 3 )
resVAL = filteredRDD.count()
```



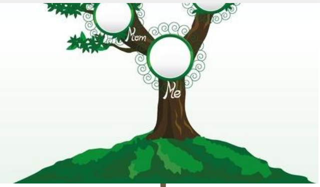
Lineage: Lazy Evaluation and Persistance

Ok, filterRDD is needed, so it is computed using mapRDD.

```
inputRDD   = sc.parallelize( [ 1, 2, 3, 4, 5 ] )
mappedRDD   = inputRDD.map( lambda elem : elem + 1 )
filteredRDD   = mappedRDD.filter( lambda elem : elem > 3 )
resVAL = filteredRDD.count()
```

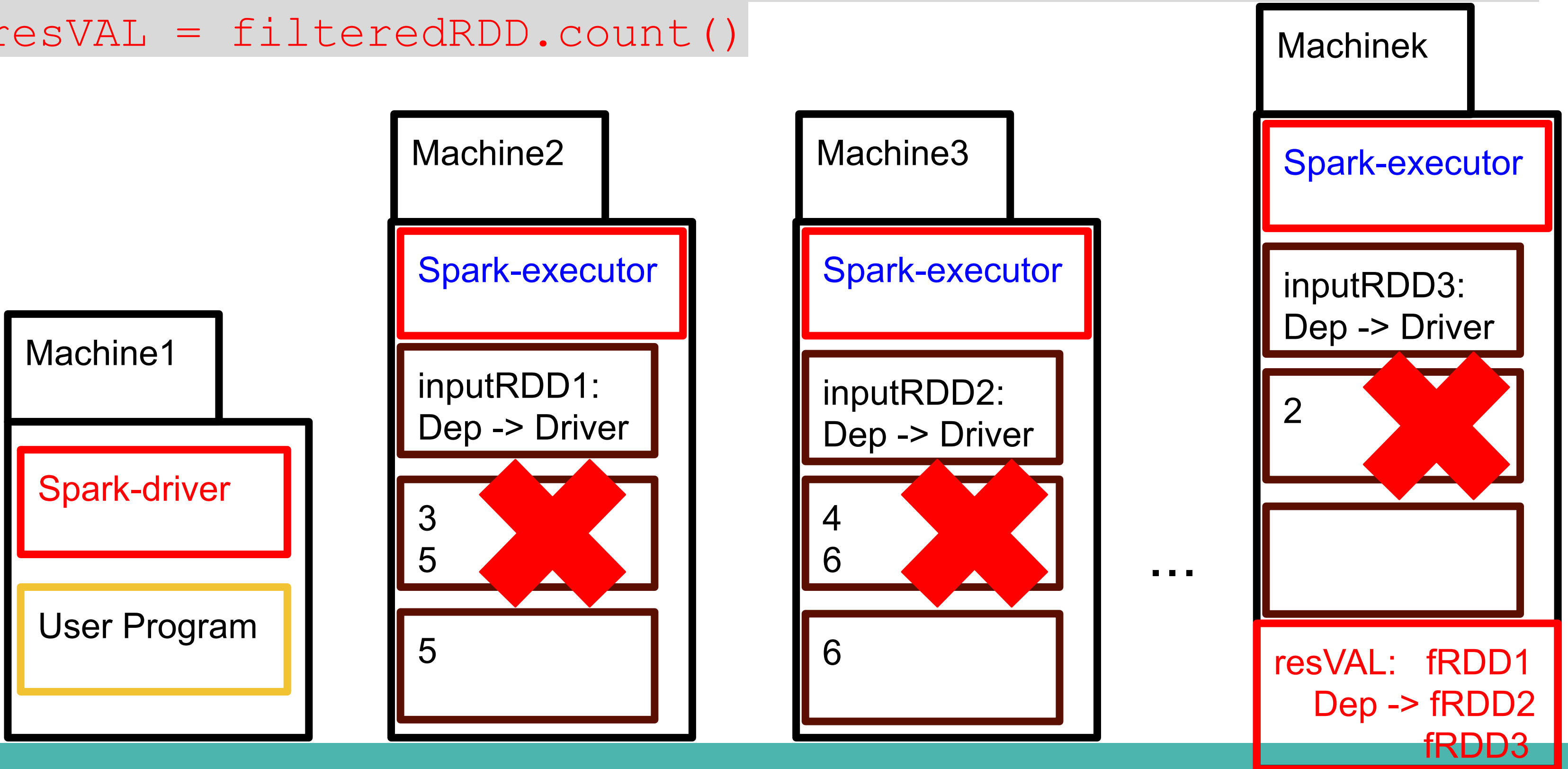


Lineage: Lazy Evaluation and Persistance

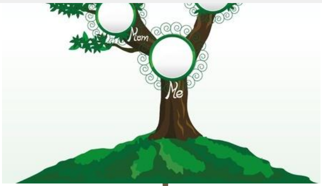


Once mappedRDD has been used, it is removed straight away!

```
inputRDD   = sc.parallelize( [ 1, 2, 3, 4, 5 ] )
mappedRDD  = inputRDD.map( lambda elem : elem + 1 )
filteredRDD  = mappedRDD.filter( lambda elem : elem > 3 )
resVAL = filteredRDD.count()
```

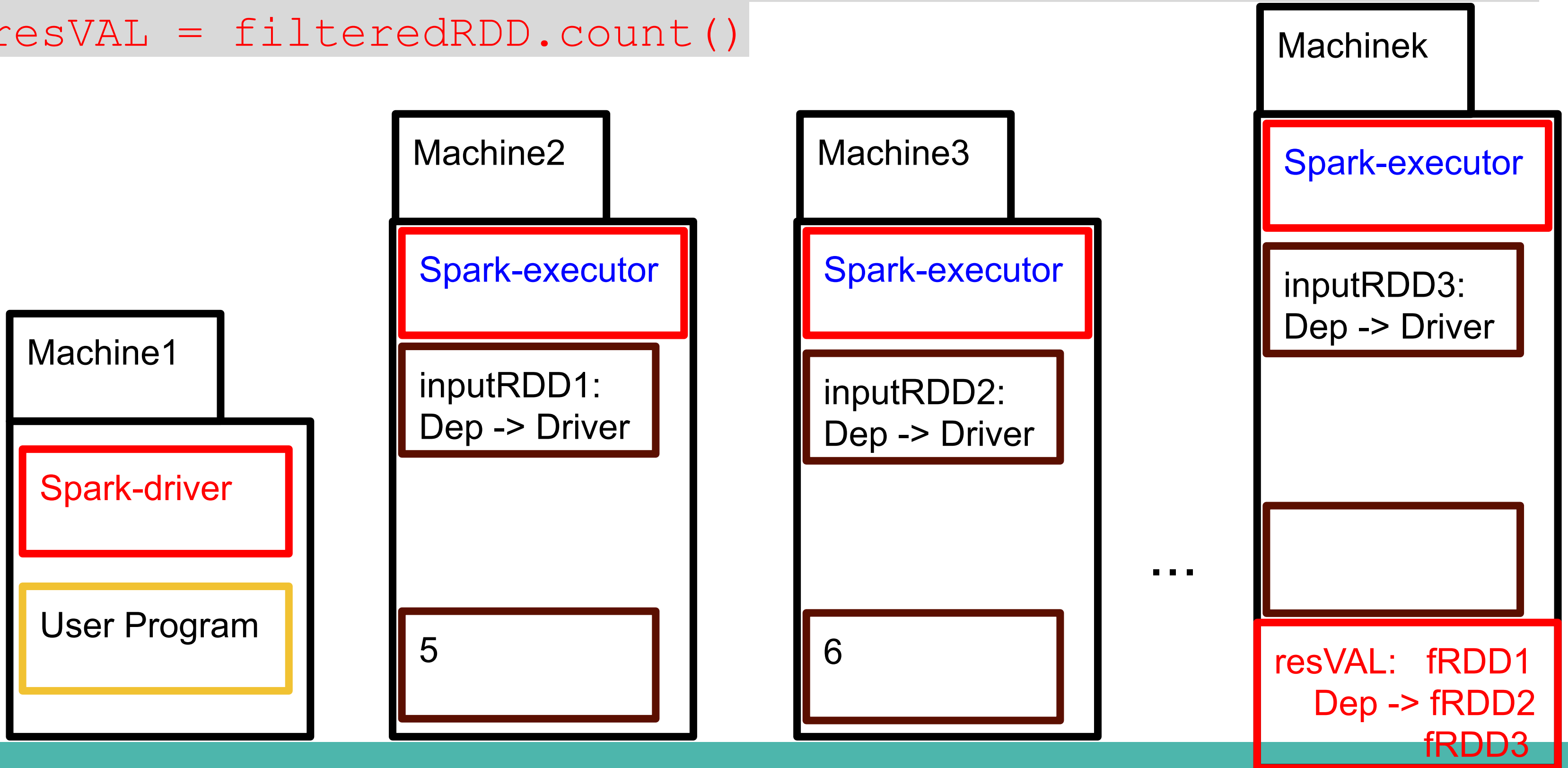


Lineage: Lazy Evaluation and Persistance



Once mappedRDD has been used, it is removed straight away!

```
inputRDD   = sc.parallelize( [ 1, 2, 3, 4, 5 ] )
mappedRDD  = inputRDD.map( lambda elem : elem + 1 )
filteredRDD  = mappedRDD.filter( lambda elem : elem > 3 )
resVAL = filteredRDD.count()
```



Machine1

Spark-driver

User Program

Machine2

Spark-executor

inputRDD1:
Dep -> Driver

5

Machine3

Spark-executor

inputRDD2:
Dep -> Driver

6

...

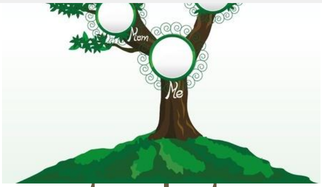
Machinek

Spark-executor

inputRDD3:
Dep -> Driver

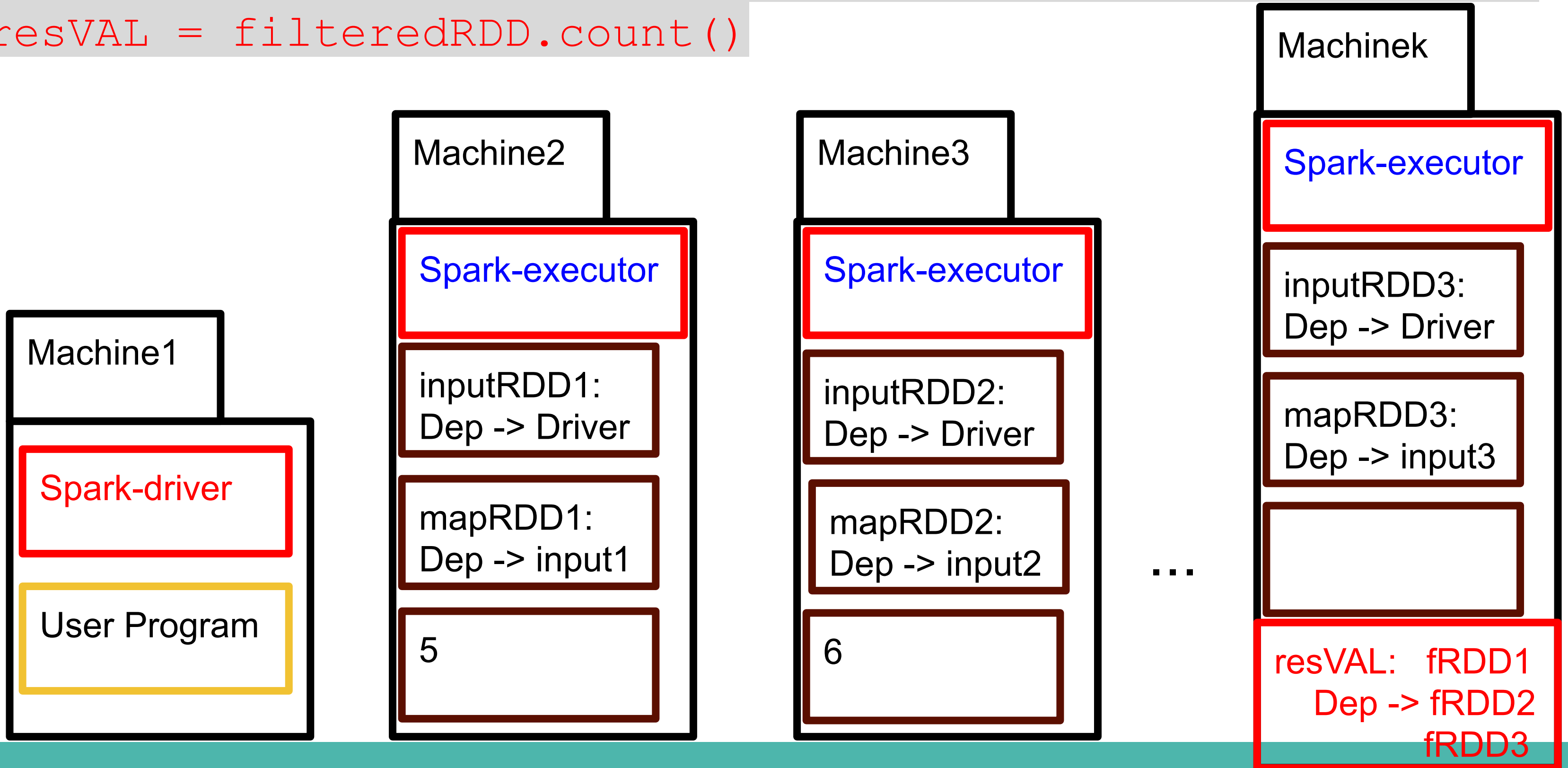
resVAL: fRDD1
Dep -> fRDD2
fRDD3

Lineage: Lazy Evaluation and Persistance

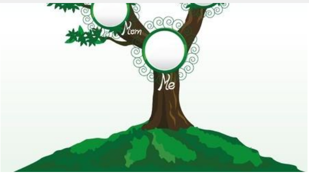


Indeed, the data of mapRDD is removed, but its lineage metadata is kept.

```
inputRDD   = sc.parallelize( [ 1, 2, 3, 4, 5 ] )
mappedRDD   = inputRDD.map( lambda elem : elem + 1 )
filteredRDD   = mappedRDD.filter( lambda elem : elem > 3 )
resVAL = filteredRDD.count()
```

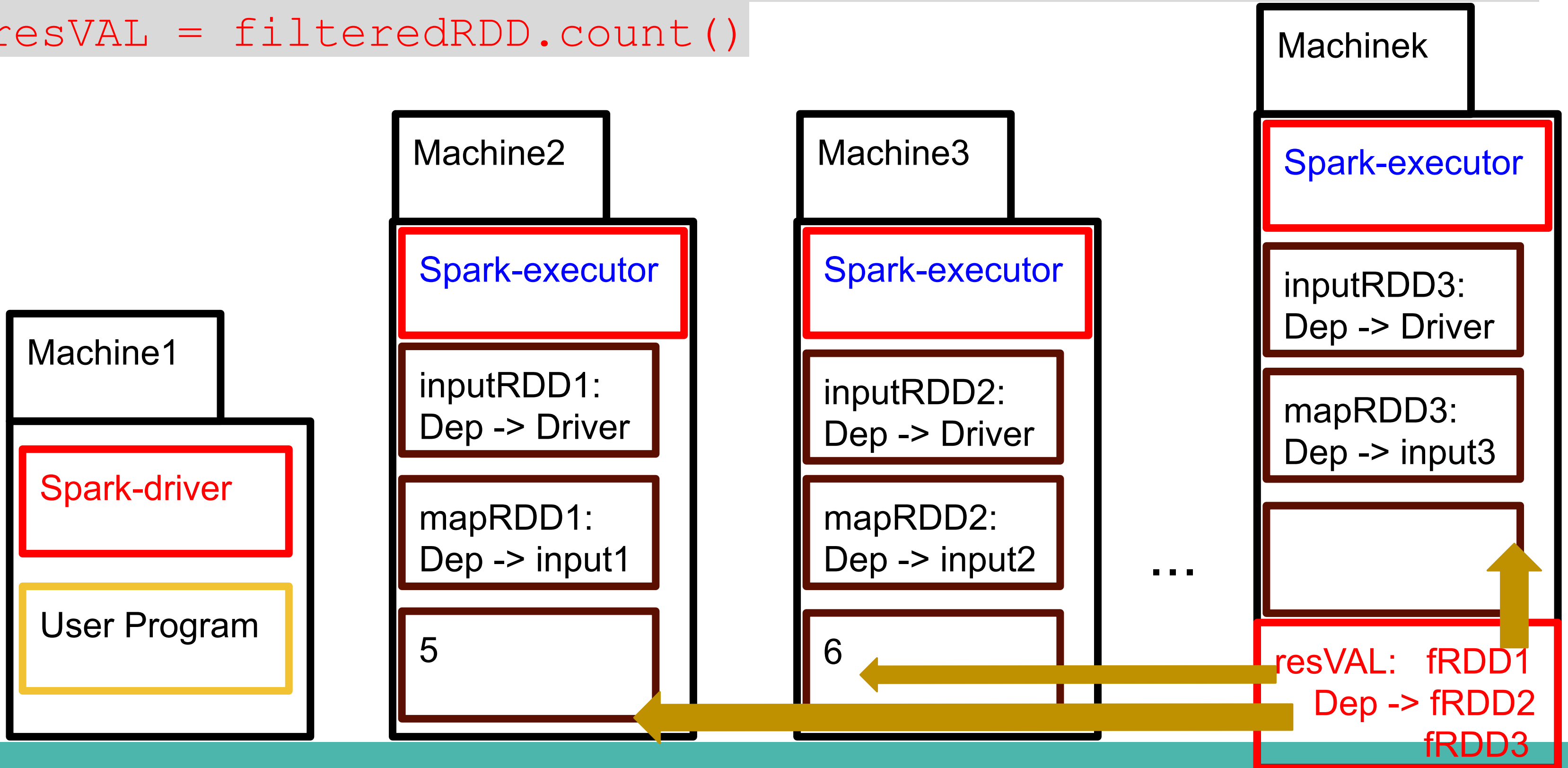


Lineage: Lazy Evaluation and Persistance

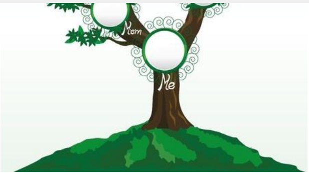


Ok, resVAL is needed, so it is computed using filterRDD.

```
inputRDD   = sc.parallelize( [ 1, 2, 3, 4, 5 ] )
mappedRDD  = inputRDD.map( lambda elem : elem + 1 )
filteredRDD  = mappedRDD.filter( lambda elem : elem > 3 )
resVAL = filteredRDD.count()
```

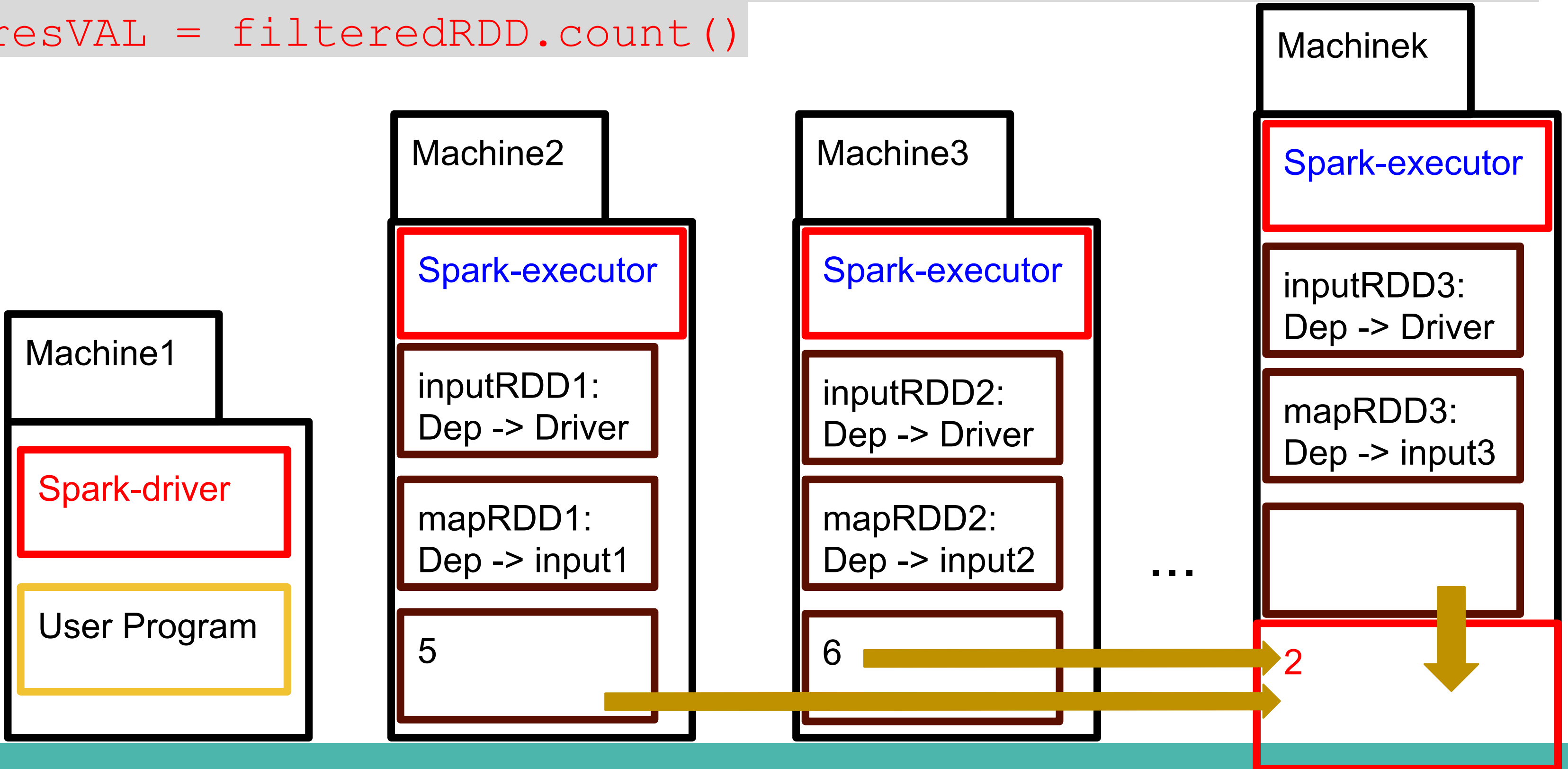


Lineage: Lazy Evaluation and Persistance

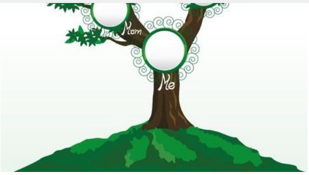


Ok, resVAL is needed, so it is computed using filterRDD.

```
inputRDD   = sc.parallelize( [ 1, 2, 3, 4, 5 ] )
mappedRDD  = inputRDD.map( lambda elem : elem + 1 )
filteredRDD  = mappedRDD.filter( lambda elem : elem > 3 )
resVAL = filteredRDD.count()
```

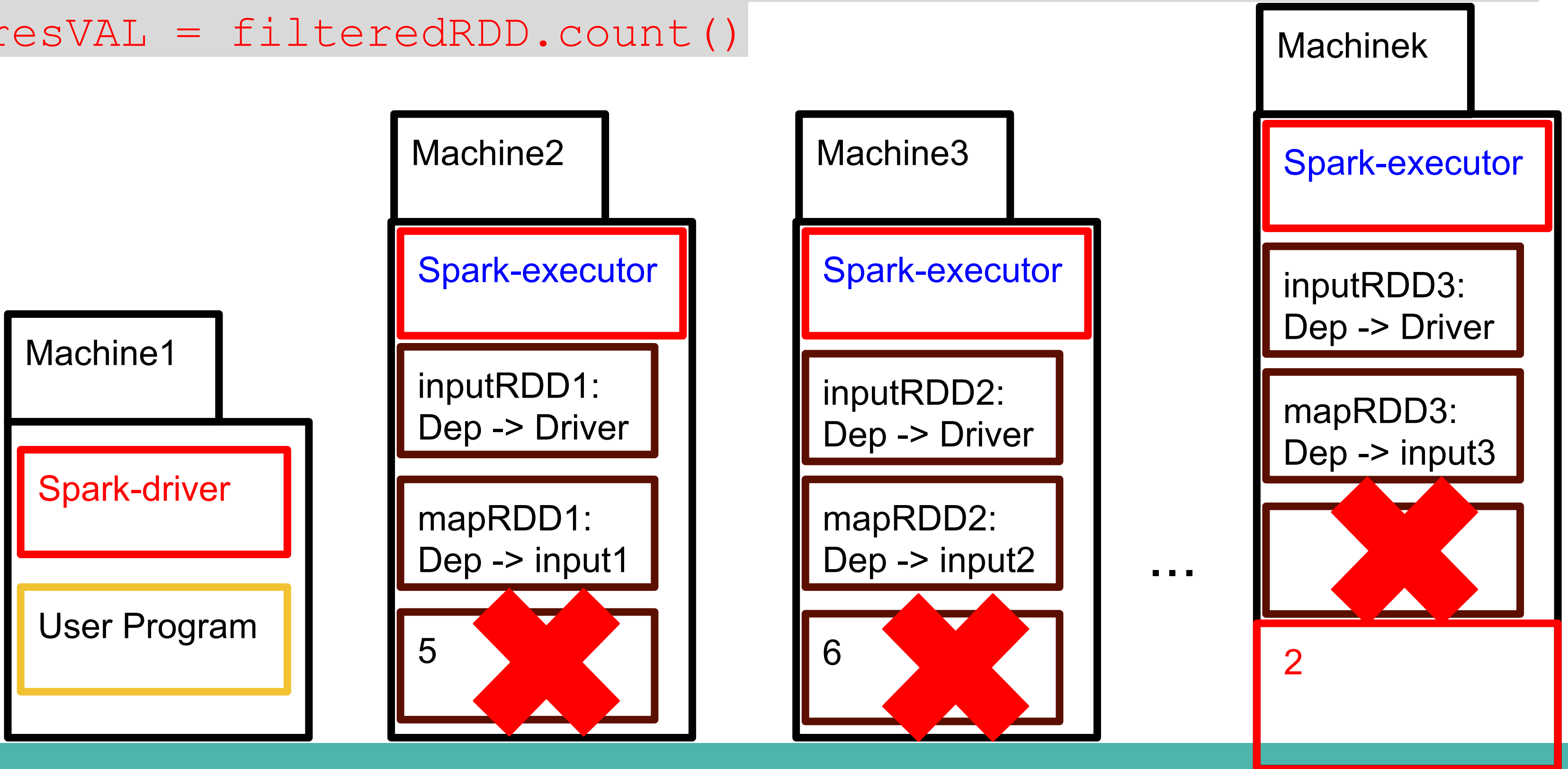


Lineage: Lazy Evaluation and Persistance

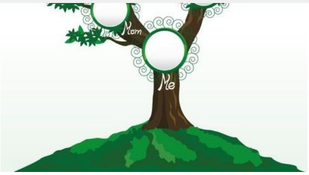


Once filterRDD has been used, it is removed straight away!

```
inputRDD  = sc.parallelize( [ 1, 2, 3, 4, 5 ] )
mappedRDD  = inputRDD.map( lambda elem : elem + 1 )
filteredRDD  = mappedRDD.filter( lambda elem : elem > 3 )
resVAL = filteredRDD.count()
```

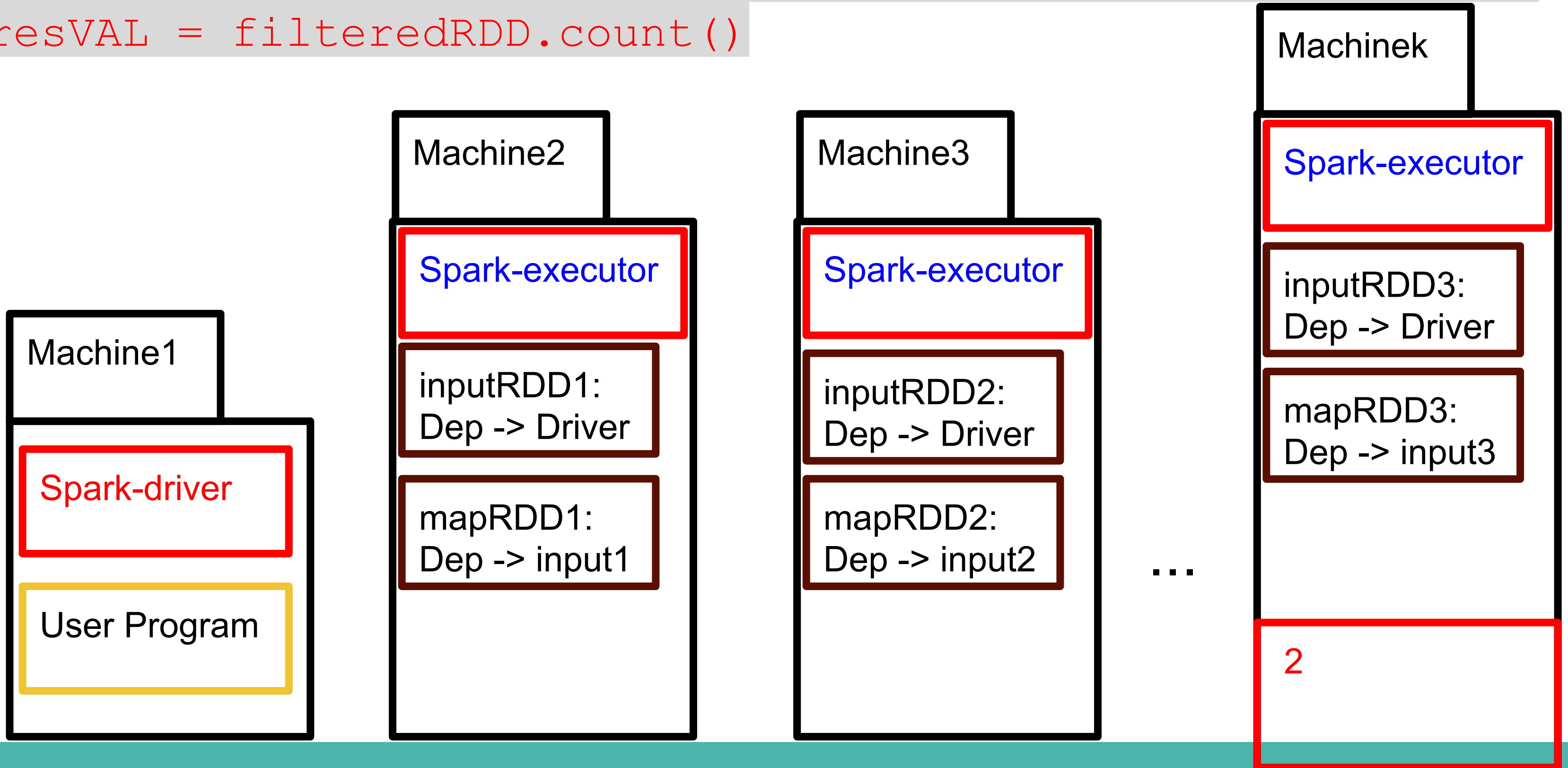


Lineage: Lazy Evaluation and Persistance



Once filterRDD has been used, it is removed straight away!

```
inputRDD   = sc.parallelize( [ 1, 2, 3, 4, 5 ] )
mappedRDD  = inputRDD.map( lambda elem : elem + 1 )
filteredRDD  = mappedRDD.filter( lambda elem : elem > 3 )
resVAL = filteredRDD.count()
```



Machine1

Spark-driver

User Program

Machine2

Spark-executor

inputRDD1:
Dep -> Driver

mapRDD1:
Dep -> input1

Machine3

Spark-executor

inputRDD2:
Dep -> Driver

mapRDD2:
Dep -> input2

...

Machinek

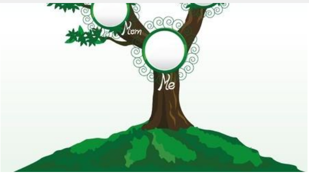
Spark-executor

inputRDD3:
Dep -> Driver

mapRDD3:
Dep -> input3

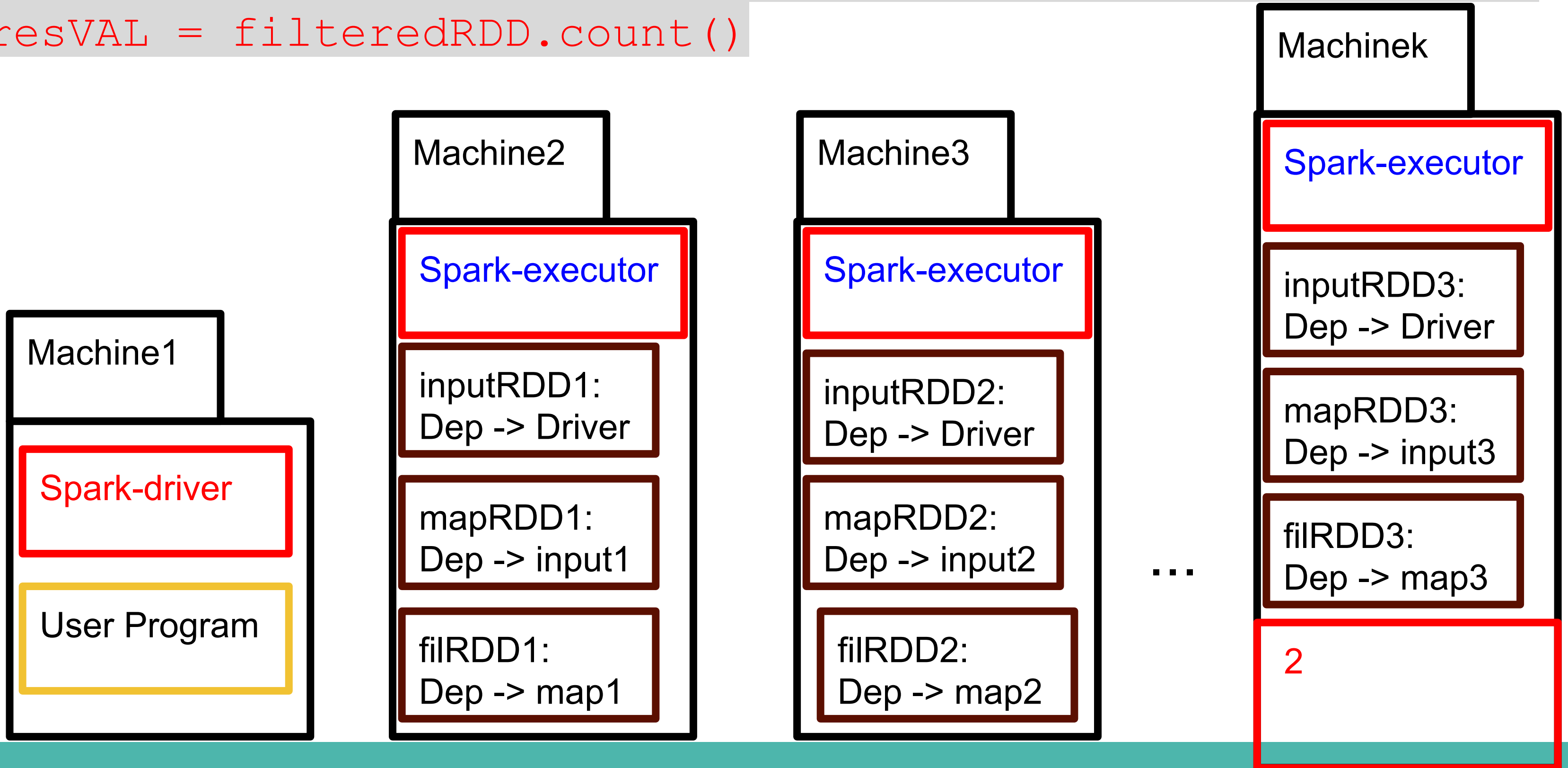
2

Lineage: Lazy Evaluation and Persistance

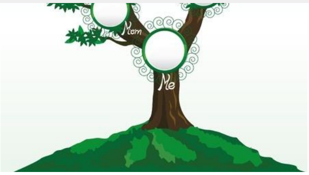


Indeed, the data of filterRDD is removed, but its lineage metadata is kept.

```
inputRDD   = sc.parallelize( [ 1, 2, 3, 4, 5 ] )
mappedRDD   = inputRDD.map( lambda elem : elem + 1 )
filteredRDD  = mappedRDD.filter( lambda elem : elem > 3 )
resVAL = filteredRDD.count()
```

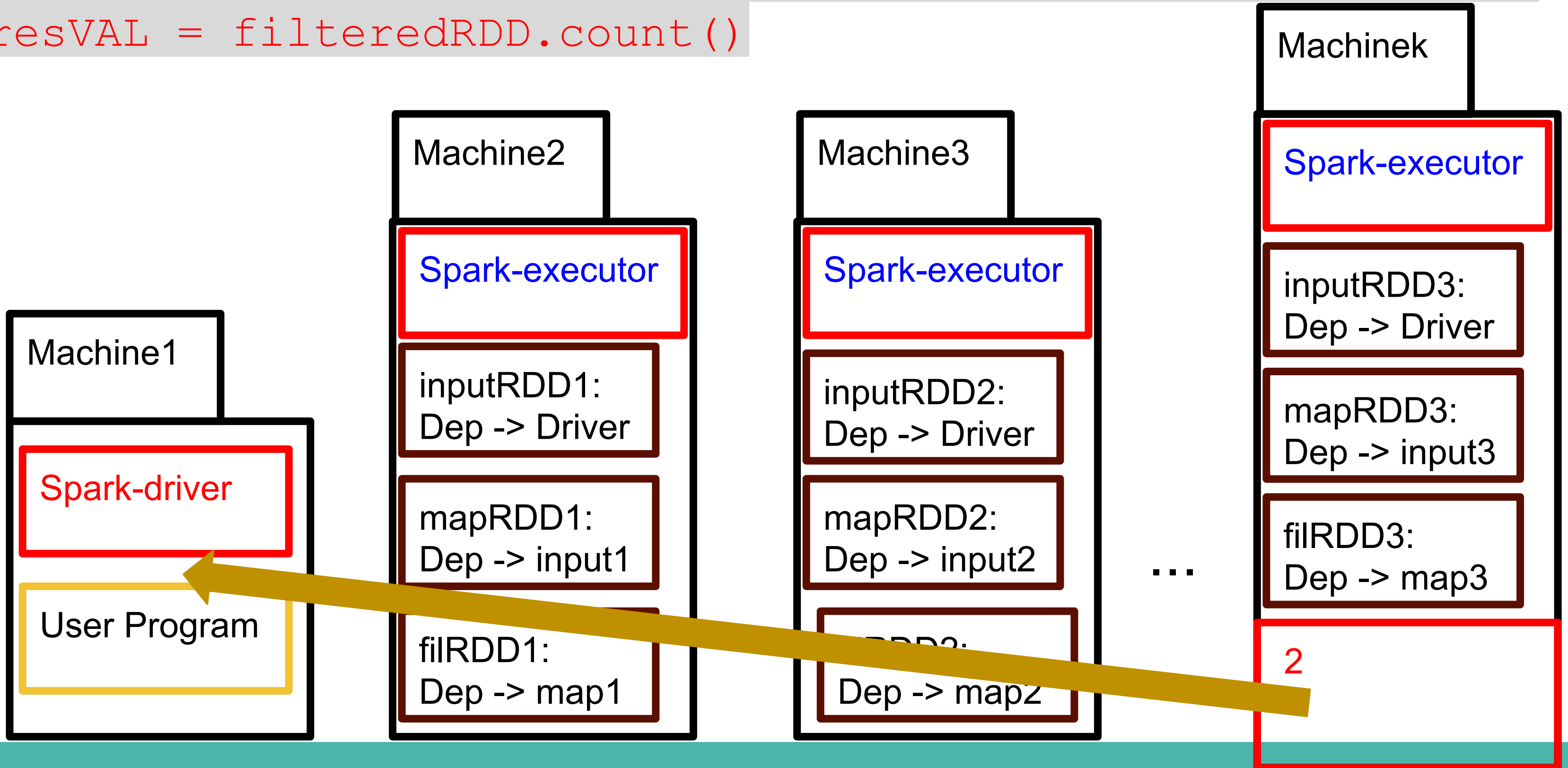


Lineage: Lazy Evaluation and Persistance



resVAL is brought back to the driver, for printing the result by the screen.

```
inputRDD   = sc.parallelize( [ 1, 2, 3, 4, 5 ] )
mappedRDD  = inputRDD.map( lambda elem : elem + 1 )
filteredRDD  = mappedRDD.filter( lambda elem : elem > 3 )
resVAL = filteredRDD.count()
```



Machine1

Spark-driver

User Program

Machine2

Spark-executor

inputRDD1: Dep -> Driver

mapRDD1: Dep -> input1

filRDD1: Dep -> map1

Machine3

Spark-executor

inputRDD2: Dep -> Driver

mapRDD2: Dep -> input2

RDD2: Dep -> map2

...

Machinek

Spark-executor

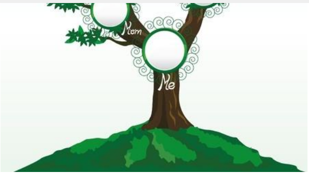
inputRDD3: Dep -> Driver

mapRDD3: Dep -> input3

filRDD3: Dep -> map3

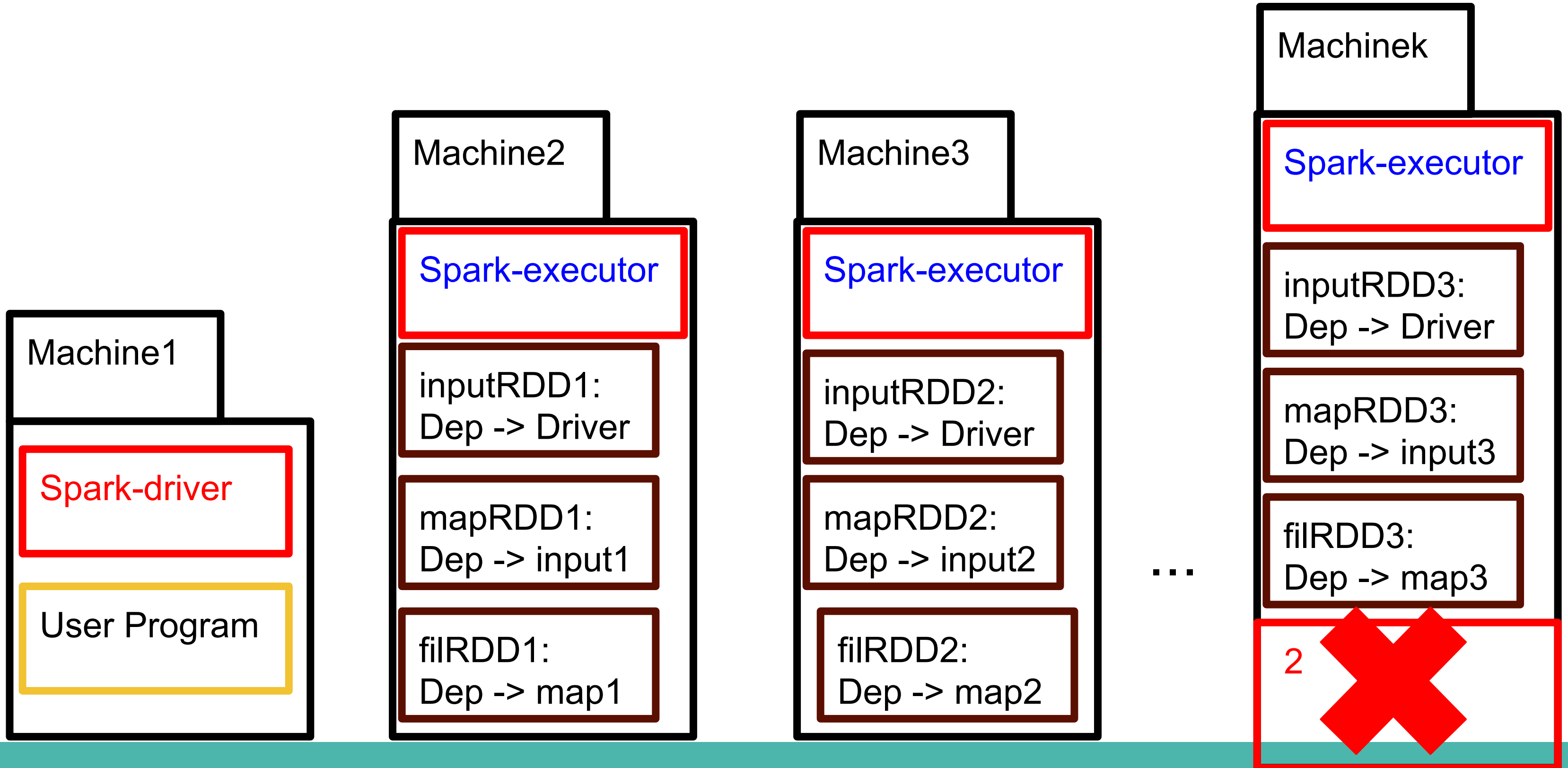
2

Lineage: Lazy Evaluation and Persistance

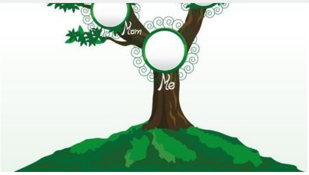


Once resVAL has been used, it is removed straight away!

```
inputRDD  = sc.parallelize( [ 1, 2, 3, 4, 5 ] )
mappedRDD  = inputRDD.map( lambda elem : elem + 1 )
filteredRDD  = mappedRDD.filter( lambda elem : elem > 3 )
resVAL = filteredRDD.count()
```

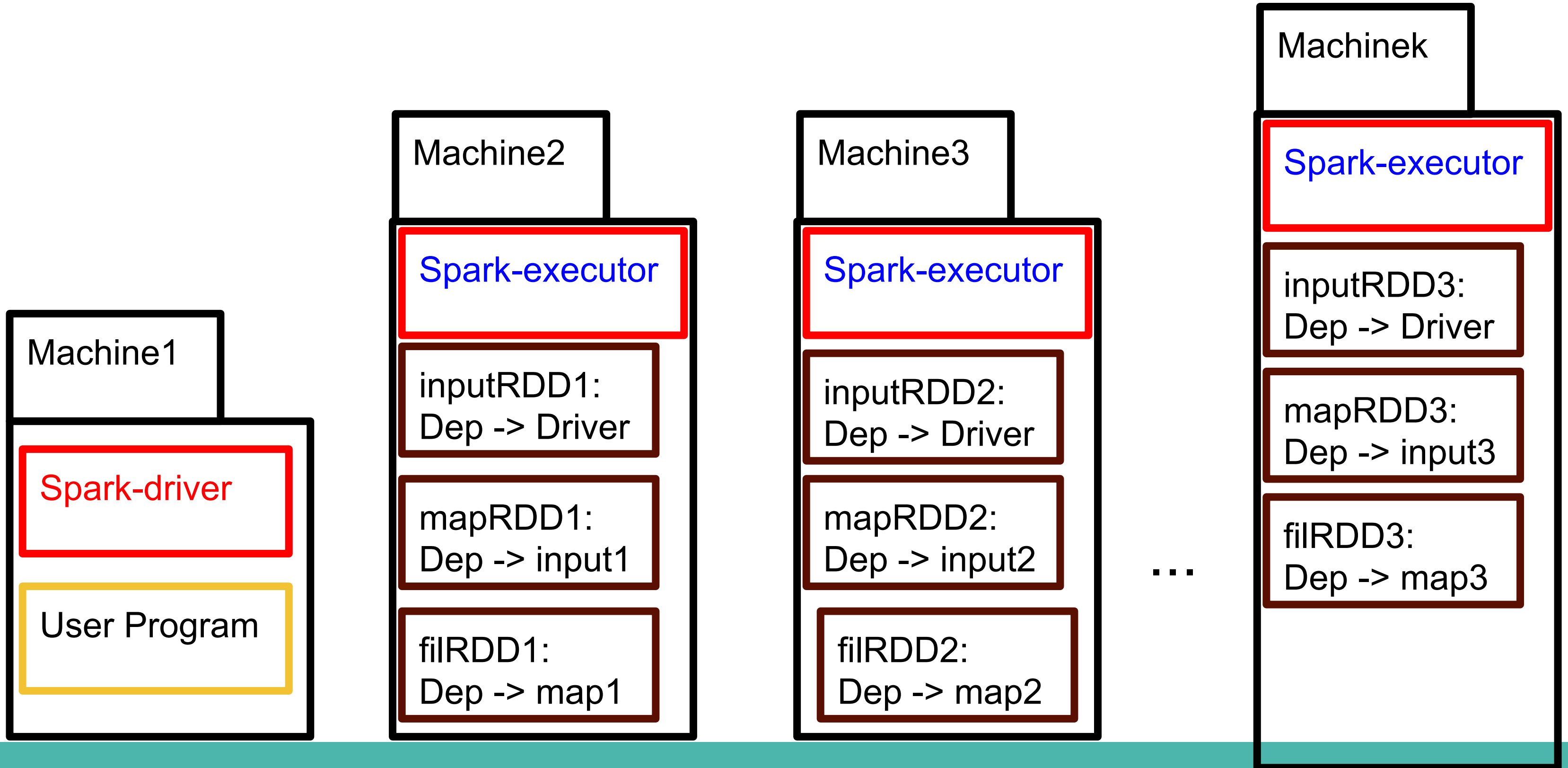


Lineage: Lazy Evaluation and Persistance

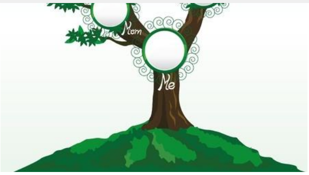


Once resVAL has been used, it is removed straight away!

```
inputRDD  = sc.parallelize( [ 1, 2, 3, 4, 5 ] )
mappedRDD  = inputRDD.map( lambda elem : elem + 1 )
filteredRDD  = mappedRDD.filter( lambda elem : elem > 3 )
resVAL = filteredRDD.count()
```

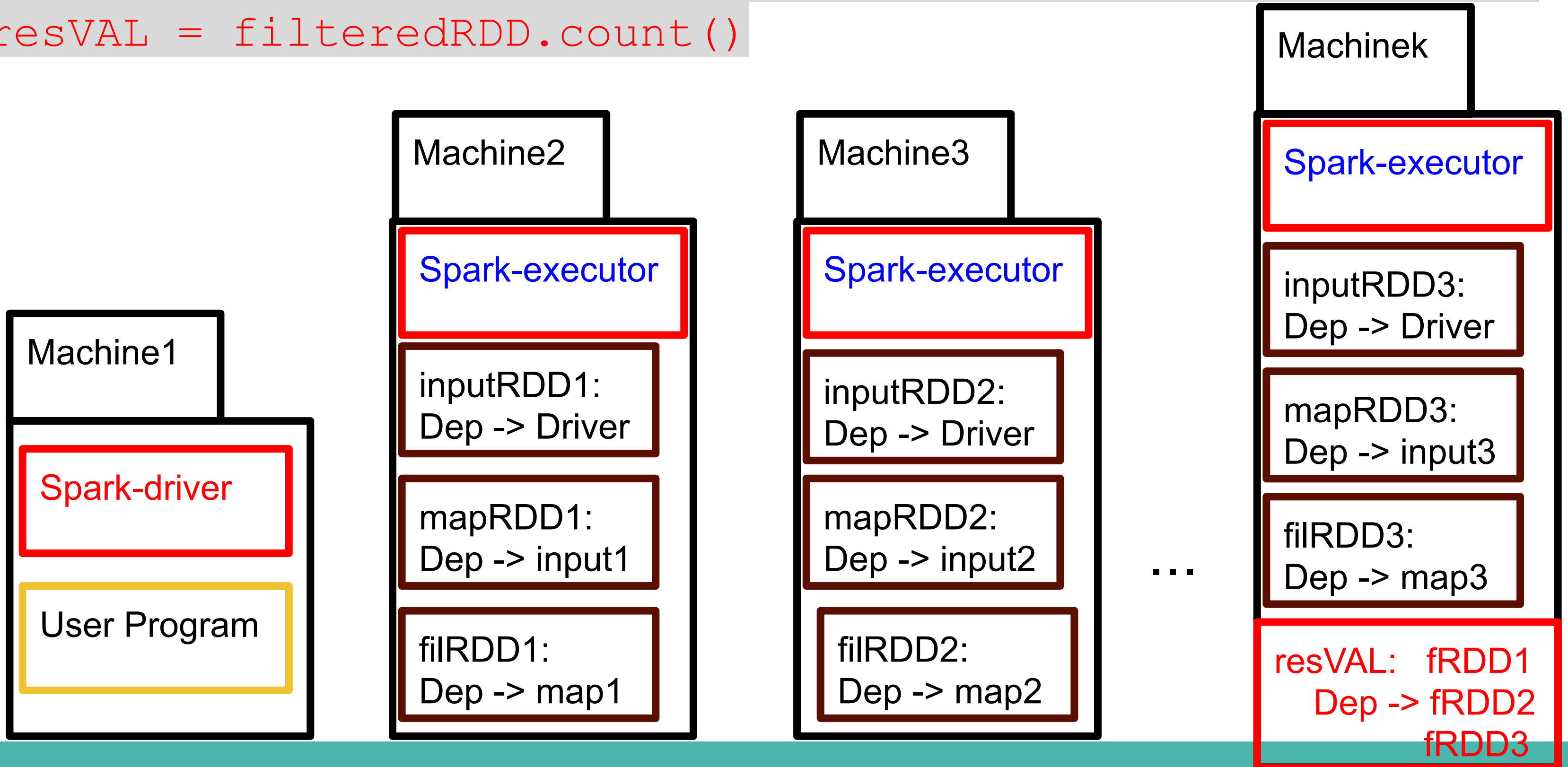


Lineage: Lazy Evaluation and Persistance



Indeed, the data of resVAL is removed, but its lineage metadata is kept.

```
inputRDD   = sc.parallelize( [ 1, 2, 3, 4, 5 ] )
mappedRDD  = inputRDD.map( lambda elem : elem + 1 )
filteredRDD  = mappedRDD.filter( lambda elem : elem > 3 )
resVAL = filteredRDD.count()
```



Lineage: Lazy Evaluation and Persistance

- The motivation for removing the RDD partitions as soon as they have been used is pretty simple:
  - We are in a Big Data environment!
    Resources are scarce, so we want to keep the memory of our **Spark Executor Processes** as free as possible!

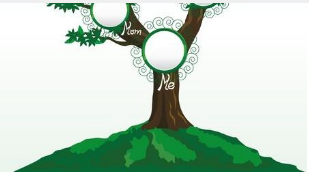
HOW'S THE BIG DATA PROJECT COMING ALONG, HOSKINS?
© D.Fletcher for CloudTweaks.com

Lineage: Lazy Evaluation and Persistance

- The motivation for removing the RDD partitions as soon as they have been used is pretty simple:
  - We are in a Big Data environment!
    Resources are scarce, so we want to keep the memory of our **Spark Executor Processes** as free as possible!

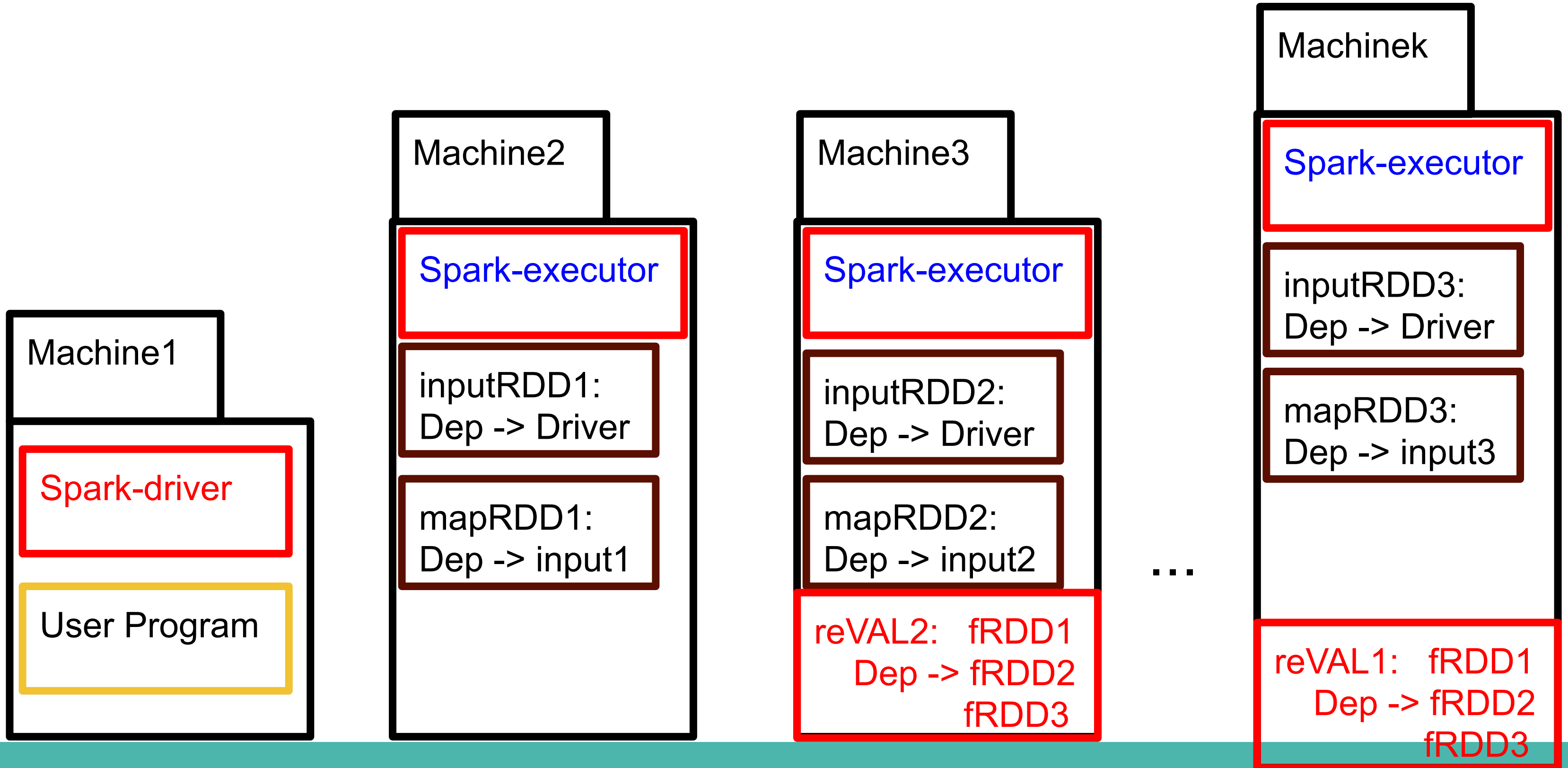
- While this idea looks wonderful on itself, it has a dark side:
  - What happens when an RDD partition is actually used twice?

Lineage: Lazy Evaluation and Persistance



Imagine the following program

```
inputRDD  = sc.parallelize( [ 1, 2, 3, 4, 5 ] )
mappedRDD  = inputRDD.map( lambda elem : elem + 1 )
resVAL1 = mappedRDD.count()
resVAL2 = mappedRDD.take(2)
```



Machine1

Spark-driver

User Program

Machine2

Spark-executor

inputRDD1:
Dep -> Driver

mapRDD1:
Dep -> input1

Machine3

Spark-executor

inputRDD2:
Dep -> Driver

mapRDD2:
Dep -> input2

reVAL2: fRDD1
Dep -> fRDD2
fRDD3

…

Machinek

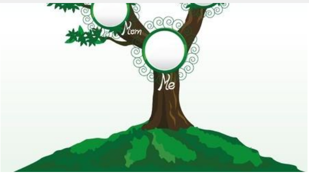
Spark-executor

inputRDD3:
Dep -> Driver

mapRDD3:
Dep -> input3

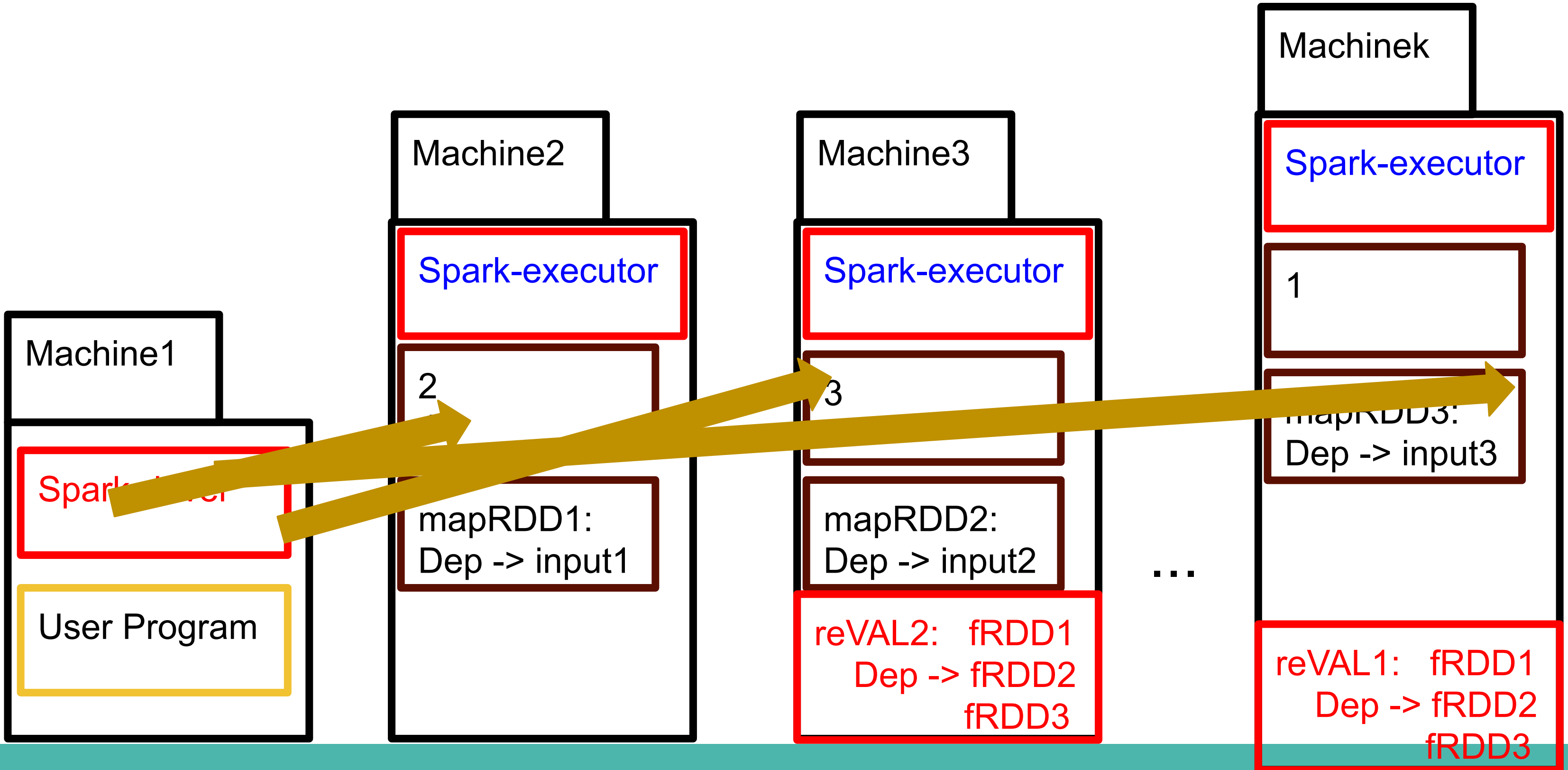
reVAL1: fRDD1
Dep -> fRDD2
fRDD3

Lineage: Lazy Evaluation and Persistance



Executing the first action count turns into computing:

```
inputRDD   = sc.parallelize( [ 1, 2, 3, 4, 5 ] )
mappedRDD   = inputRDD.map( lambda elem : elem + 1 )
resVAL1 = mappedRDD.count()
resVAL2 = mappedRDD.take(2)
```



Machine1

Spark-driver

User Program

Machine2

Spark-executor

2

mapRDD1:
Dep -> input1

Machine3

Spark-executor

3

mapRDD2:
Dep -> input2

reVAL2: fRDD1
Dep -> fRDD2
fRDD3

...

Machinek

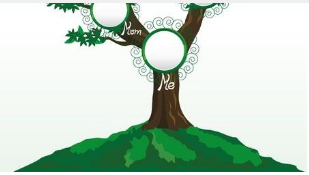
Spark-executor

1

mapRDD3:
Dep -> input3

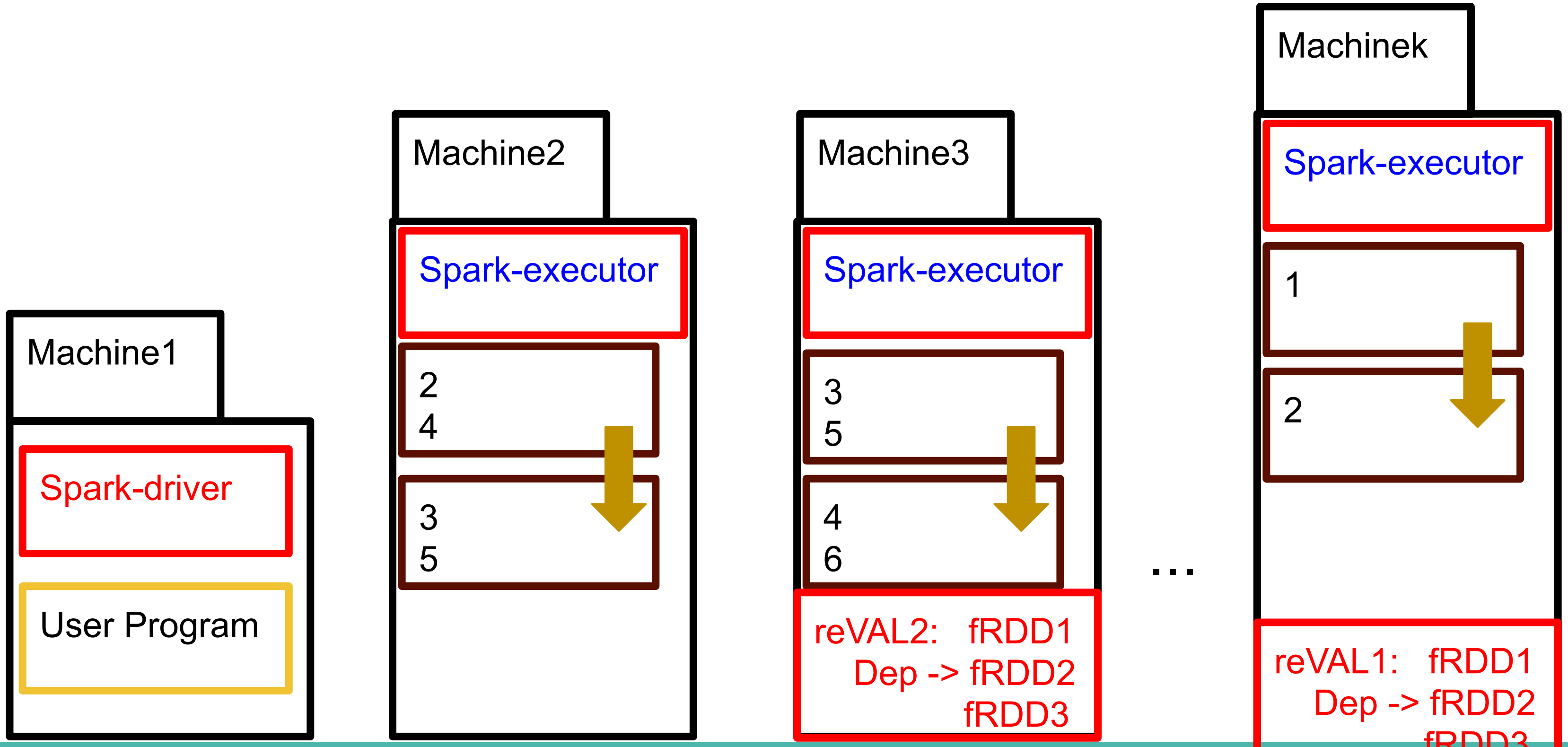
reVAL1: fRDD1
Dep -> fRDD2
fRDD3

Lineage: Lazy Evaluation and Persistance

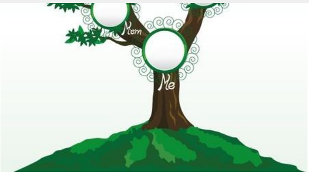


Executing the first action count turns into computing:

```
inputRDD  = sc.parallelize( [ 1, 2, 3, 4, 5 ] )
mappedRDD  = inputRDD.map( lambda elem : elem + 1 )
resVAL1 = mappedRDD.count()
resVAL2 = mappedRDD.take(2)
```

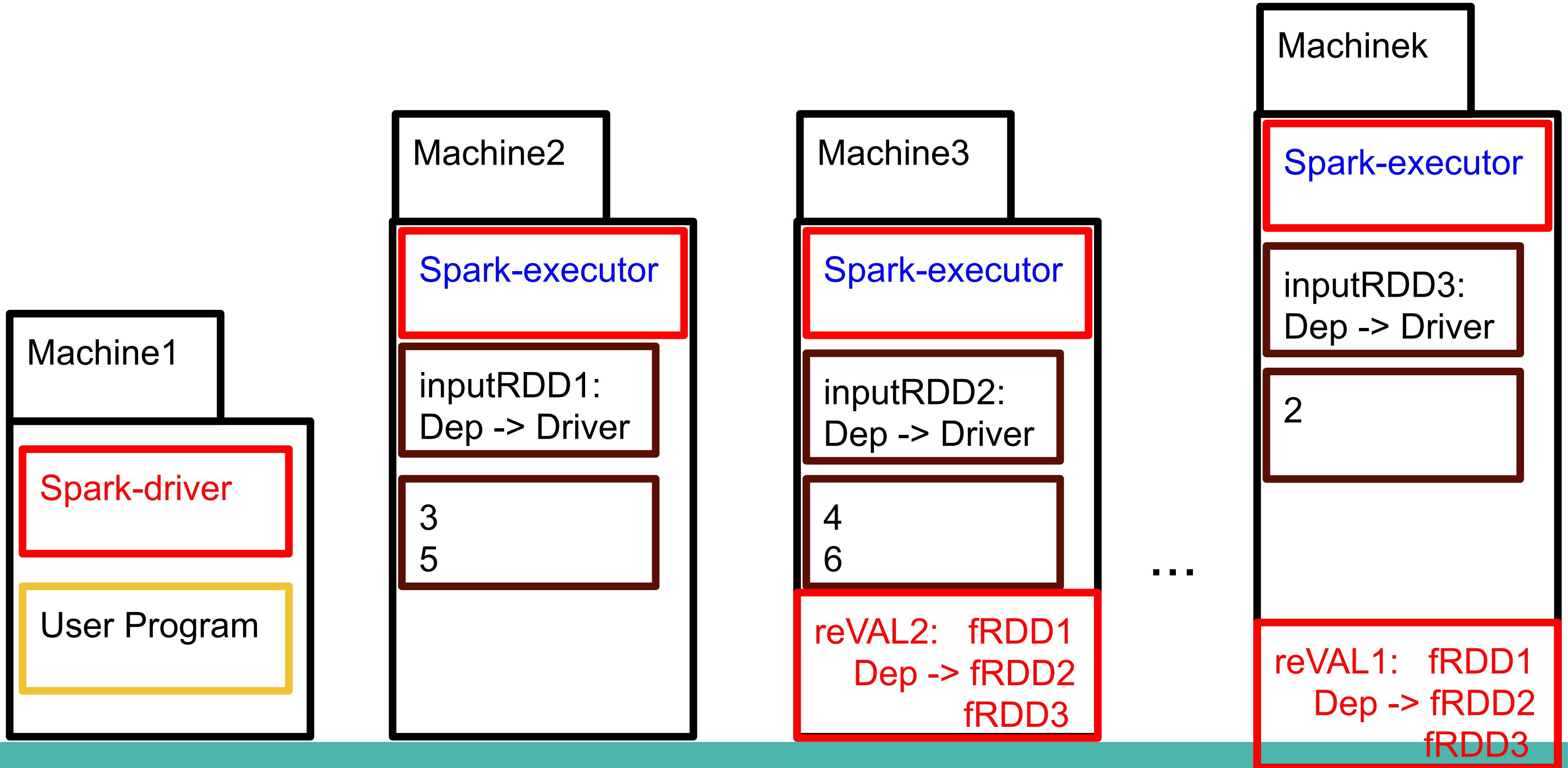


Lineage: Lazy Evaluation and Persistance



Executing the first action count turns into computing:

```
inputRDD  = sc.parallelize( [ 1, 2, 3, 4, 5 ] )
mappedRDD  = inputRDD.map( lambda elem : elem + 1 )
resVAL1 = mappedRDD.count()
resVAL2 = mappedRDD.take(2)
```



Machine1

Spark-driver

User Program

Machine2

Spark-executor

inputRDD1:
Dep -> Driver

3
5

Machine3

Spark-executor

inputRDD2:
Dep -> Driver

4
6

reVAL2: fRDD1
Dep -> fRDD2
fRDD3

…

Machinek

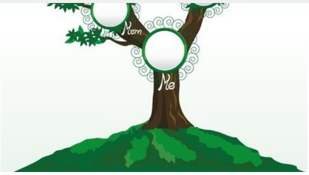
Spark-executor

inputRDD3:
Dep -> Driver

2

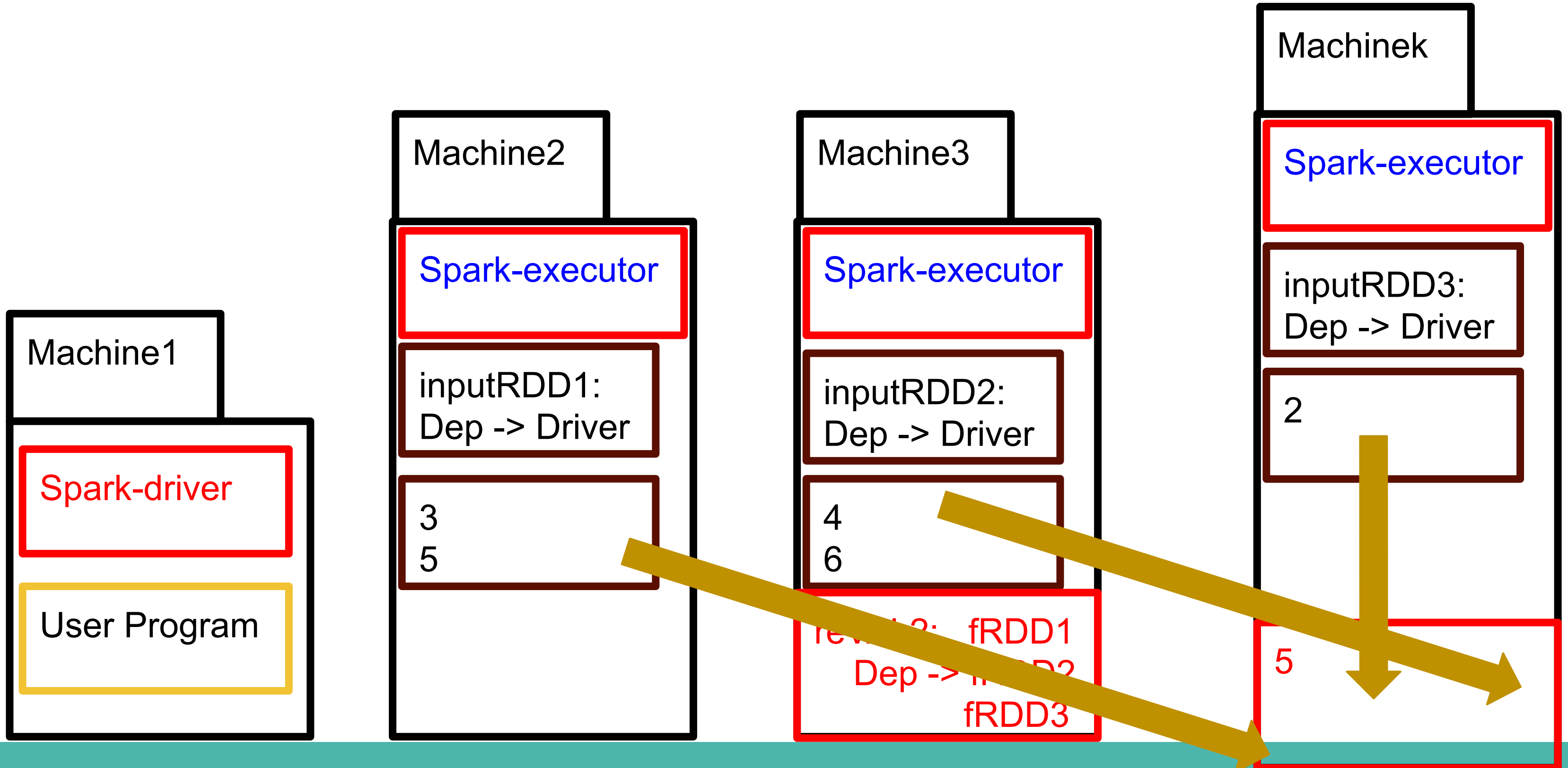
reVAL1: fRDD1
Dep -> fRDD2
fRDD3

Lineage: Lazy Evaluation and Persistance

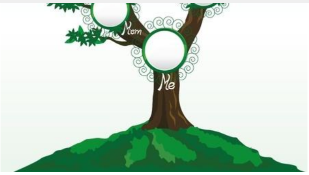


Executing the first action count turns into computing:

```
inputRDD  = sc.parallelize( [ 1, 2, 3, 4, 5 ] )
mappedRDD  = inputRDD.map( lambda elem : elem + 1 )
resVAL1 = mappedRDD.count()
resVAL2 = mappedRDD.take(2)
```

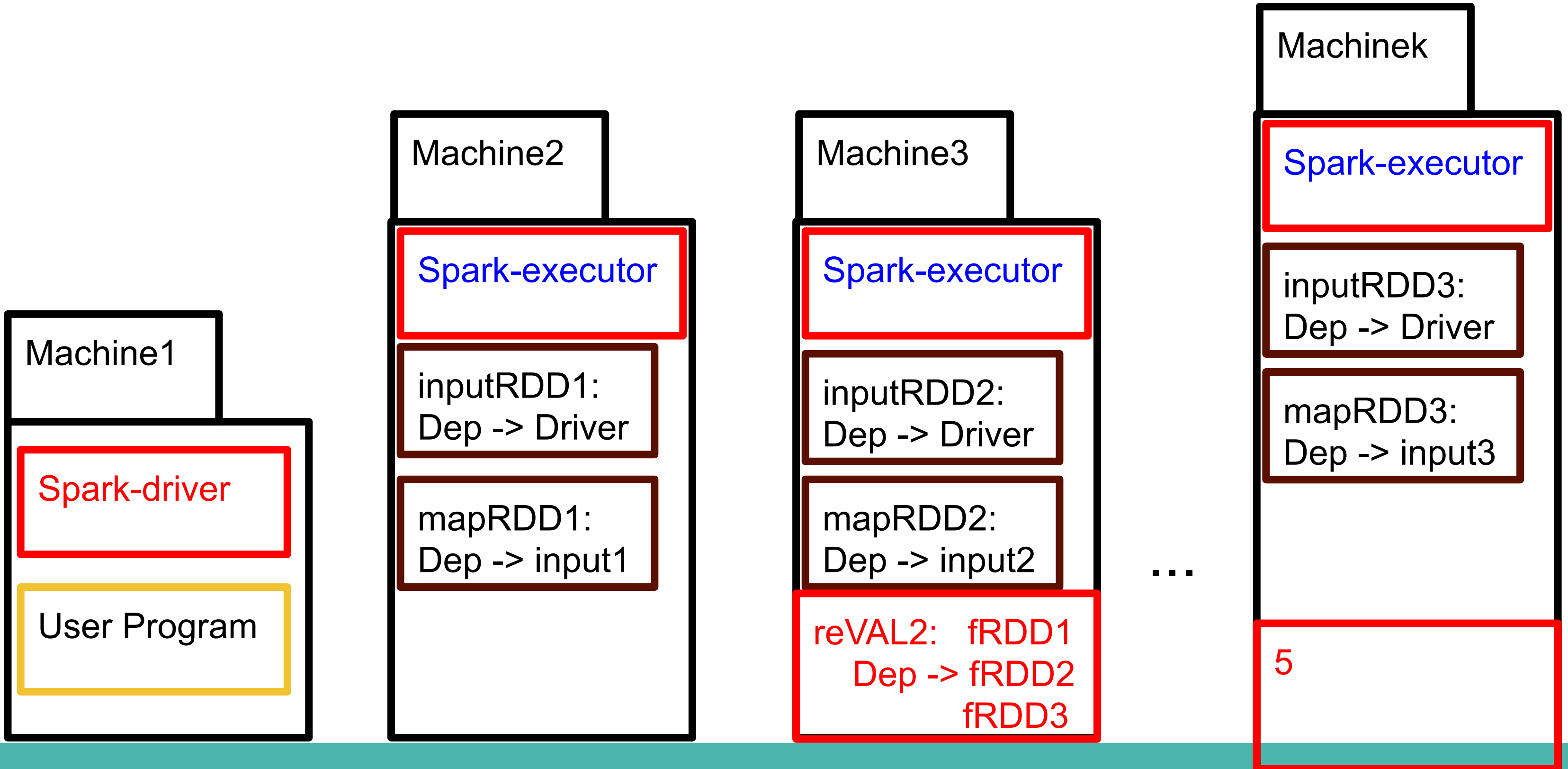


Lineage: Lazy Evaluation and Persistance

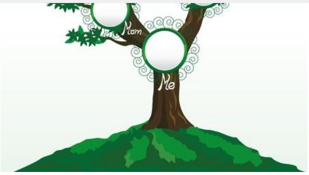


Executing the first action count turns into computing:

```
inputRDD  = sc.parallelize( [ 1, 2, 3, 4, 5 ] )
mappedRDD  = inputRDD.map( lambda elem : elem + 1 )
resVAL1 = mappedRDD.count()
resVAL2 = mappedRDD.take(2)
```

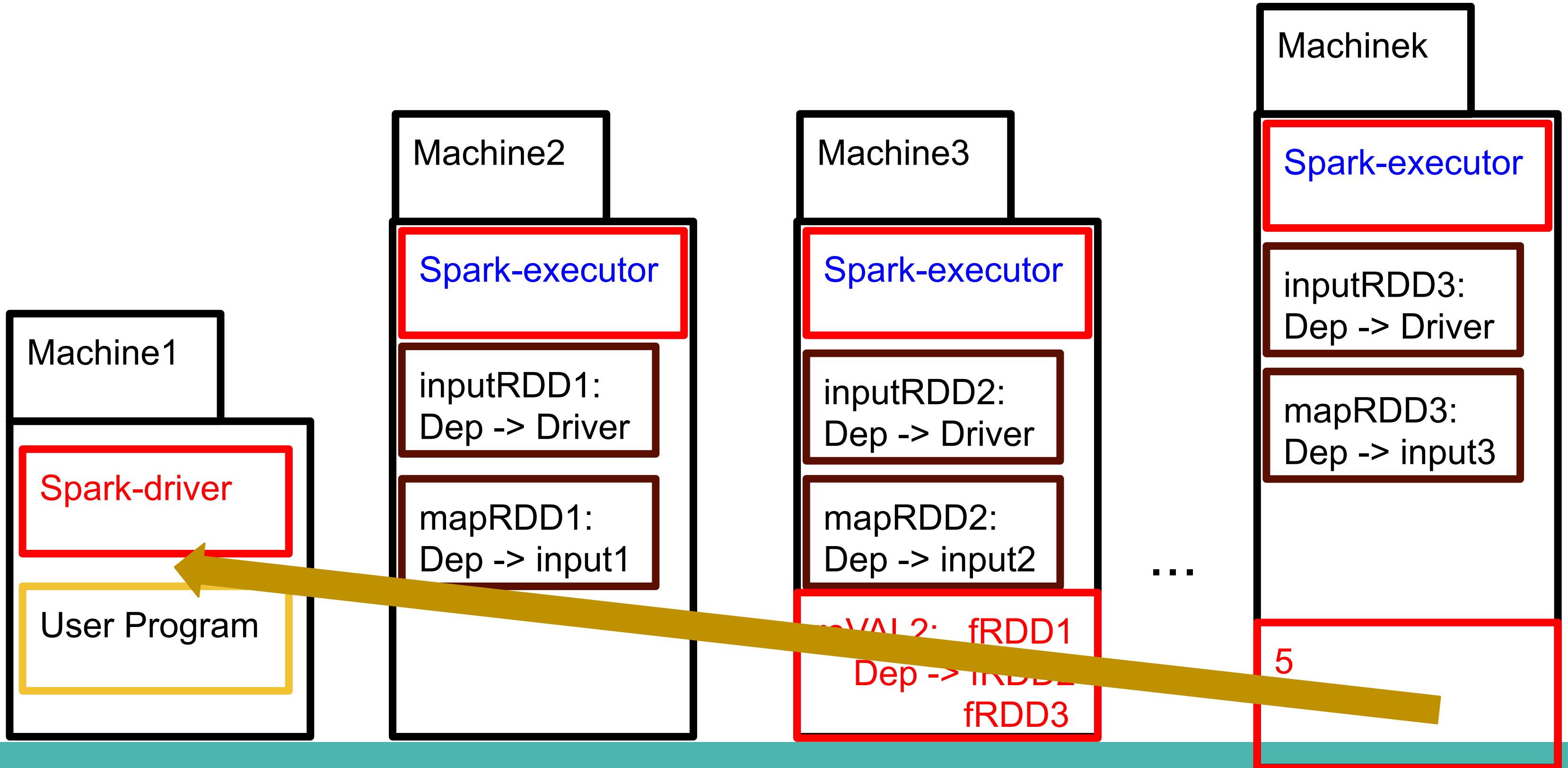


Lineage: Lazy Evaluation and Persistance

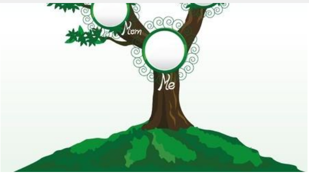


Executing the first action count turns into computing:

```
inputRDD  = sc.parallelize( [ 1, 2, 3, 4, 5 ] )
mappedRDD  = inputRDD.map( lambda elem : elem + 1 )
resVAL1 = mappedRDD.count()
resVAL2 = mappedRDD.take(2)
```

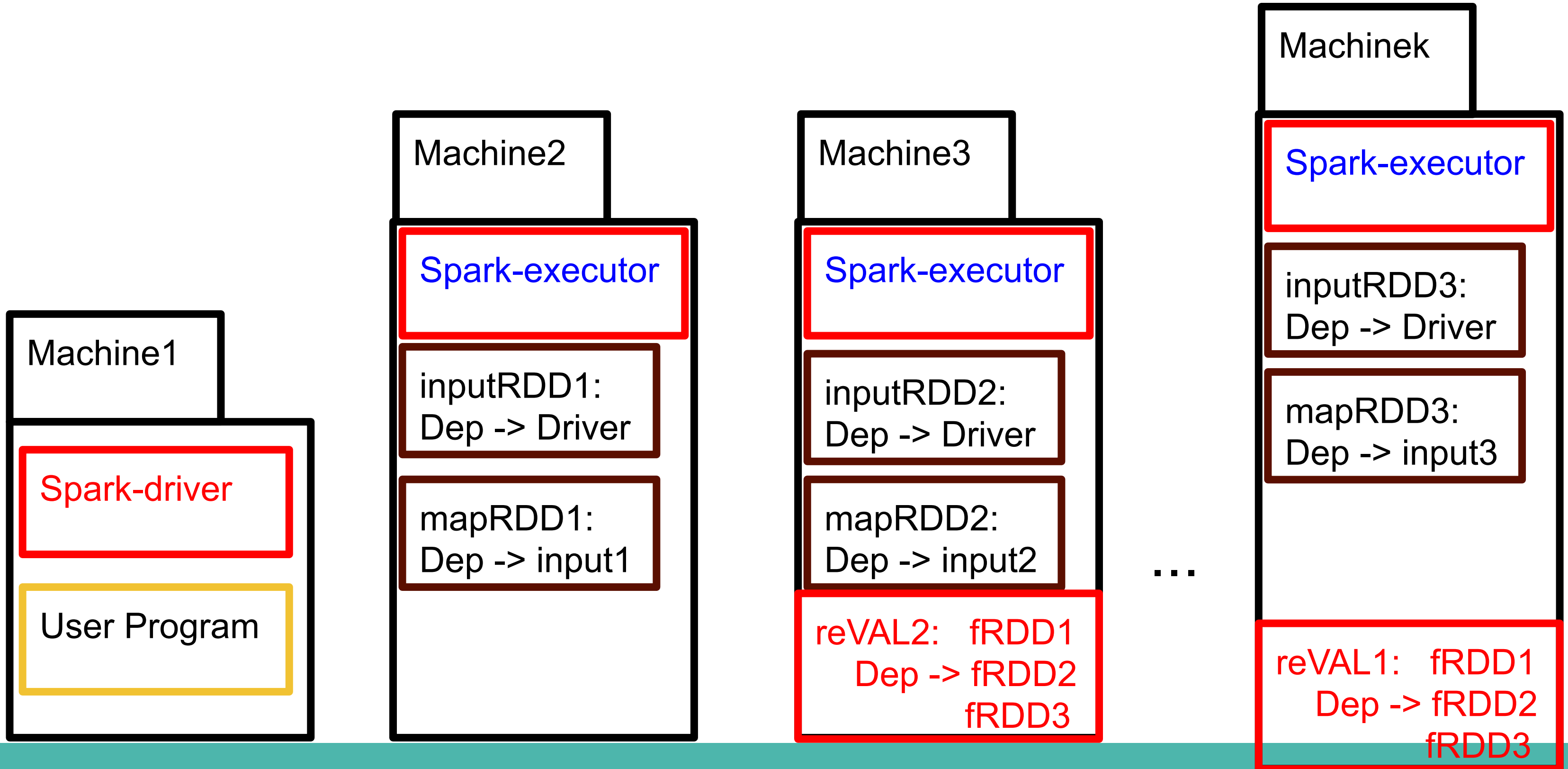


Lineage: Lazy Evaluation and Persistance



Executing the first action count turns into computing:

```
inputRDD  = sc.parallelize( [ 1, 2, 3, 4, 5 ] )
mappedRDD  = inputRDD.map( lambda elem : elem + 1 )
resVAL1 = mappedRDD.count()
resVAL2 = mappedRDD.take(2)
```



Machine1

Spark-driver

User Program

Machine2

Spark-executor

inputRDD1:
Dep -> Driver

mapRDD1:
Dep -> input1

Machine3

Spark-executor

inputRDD2:
Dep -> Driver

mapRDD2:
Dep -> input2

reVAL2: fRDD1
Dep -> fRDD2
fRDD3

…

Machinek

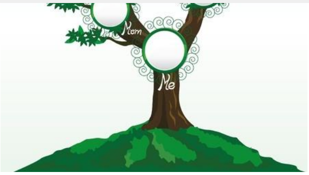
Spark-executor

inputRDD3:
Dep -> Driver

mapRDD3:
Dep -> input3

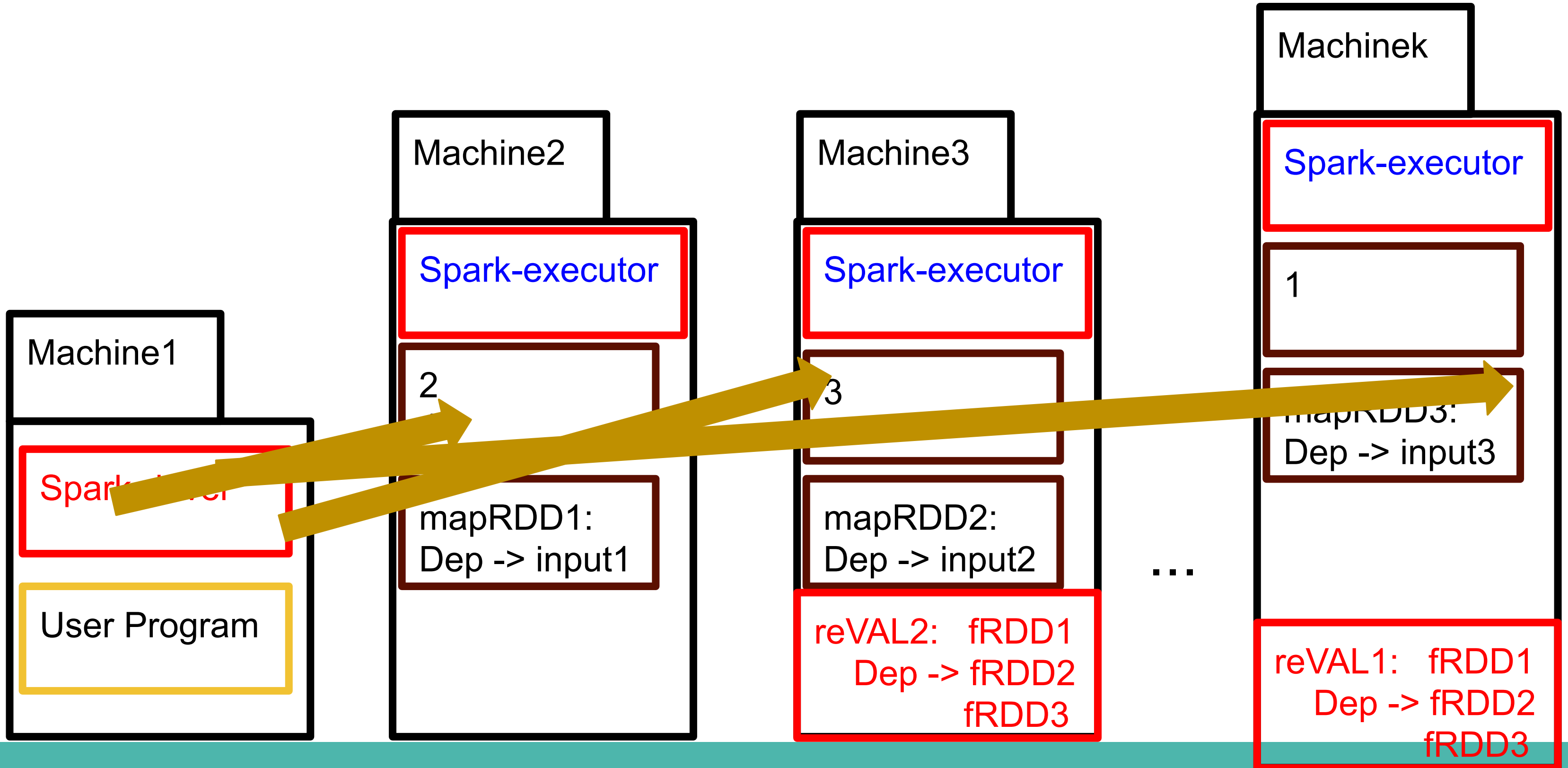
reVAL1: fRDD1
Dep -> fRDD2
fRDD3

Lineage: Lazy Evaluation and Persistance

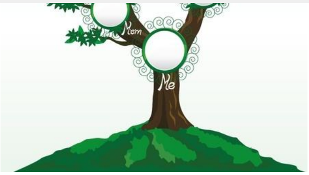


Executing the second action take turns into computing:

```
inputRDD  = sc.parallelize( [ 1, 2, 3, 4, 5 ] )
mappedRDD  = inputRDD.map( lambda elem : elem + 1 )
resVAL1 = mappedRDD.count()
resVAL2 = mappedRDD.take(2)
```

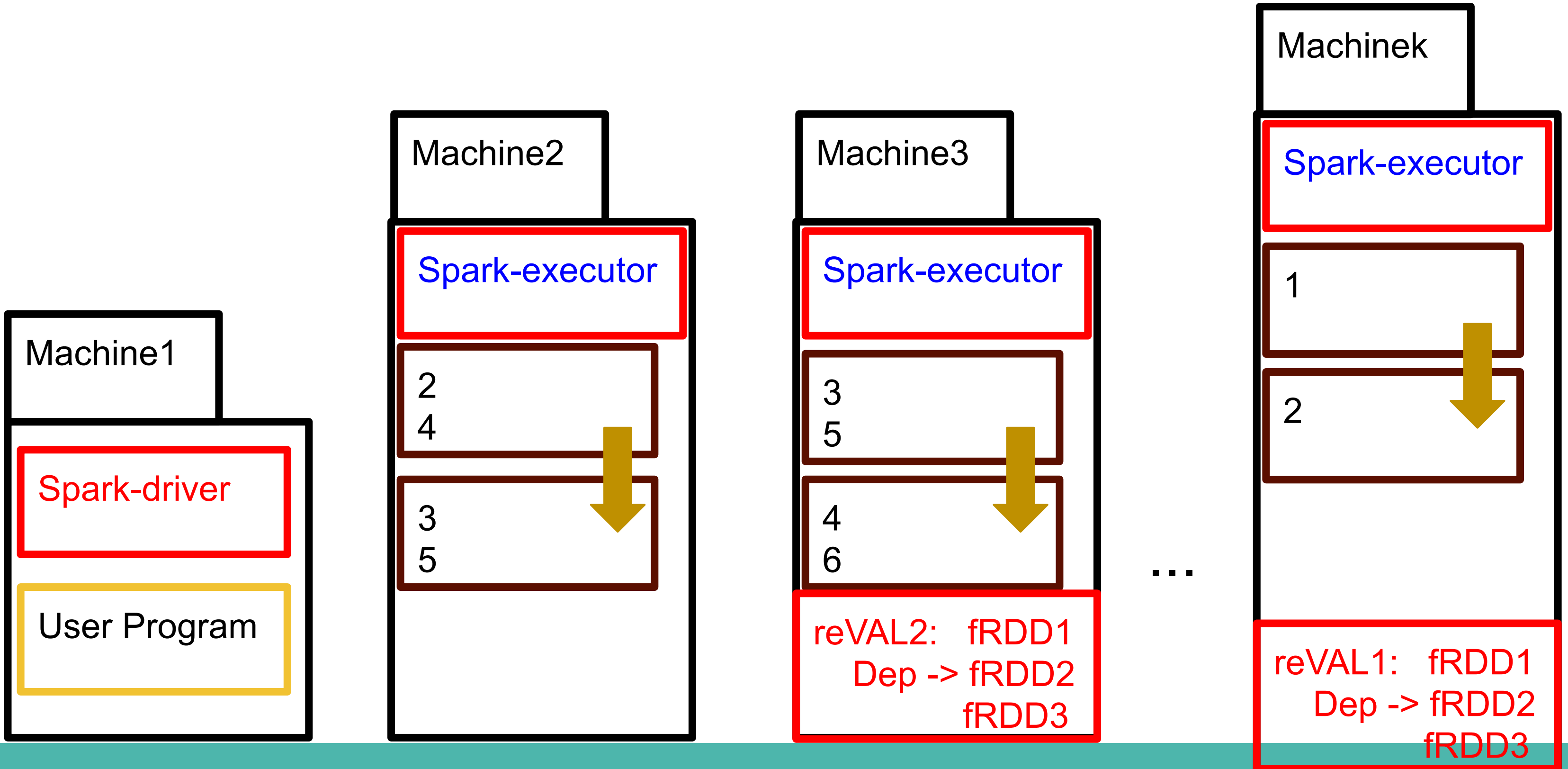


Lineage: Lazy Evaluation and Persistance

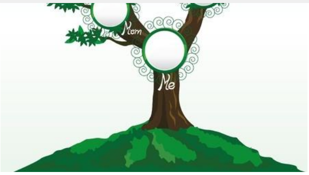


Executing the second action take turns into computing:

```
inputRDD  = sc.parallelize( [ 1, 2, 3, 4, 5 ] )
mappedRDD  = inputRDD.map( lambda elem : elem + 1 )
resVAL1 = mappedRDD.count()
resVAL2 = mappedRDD.take(2)
```

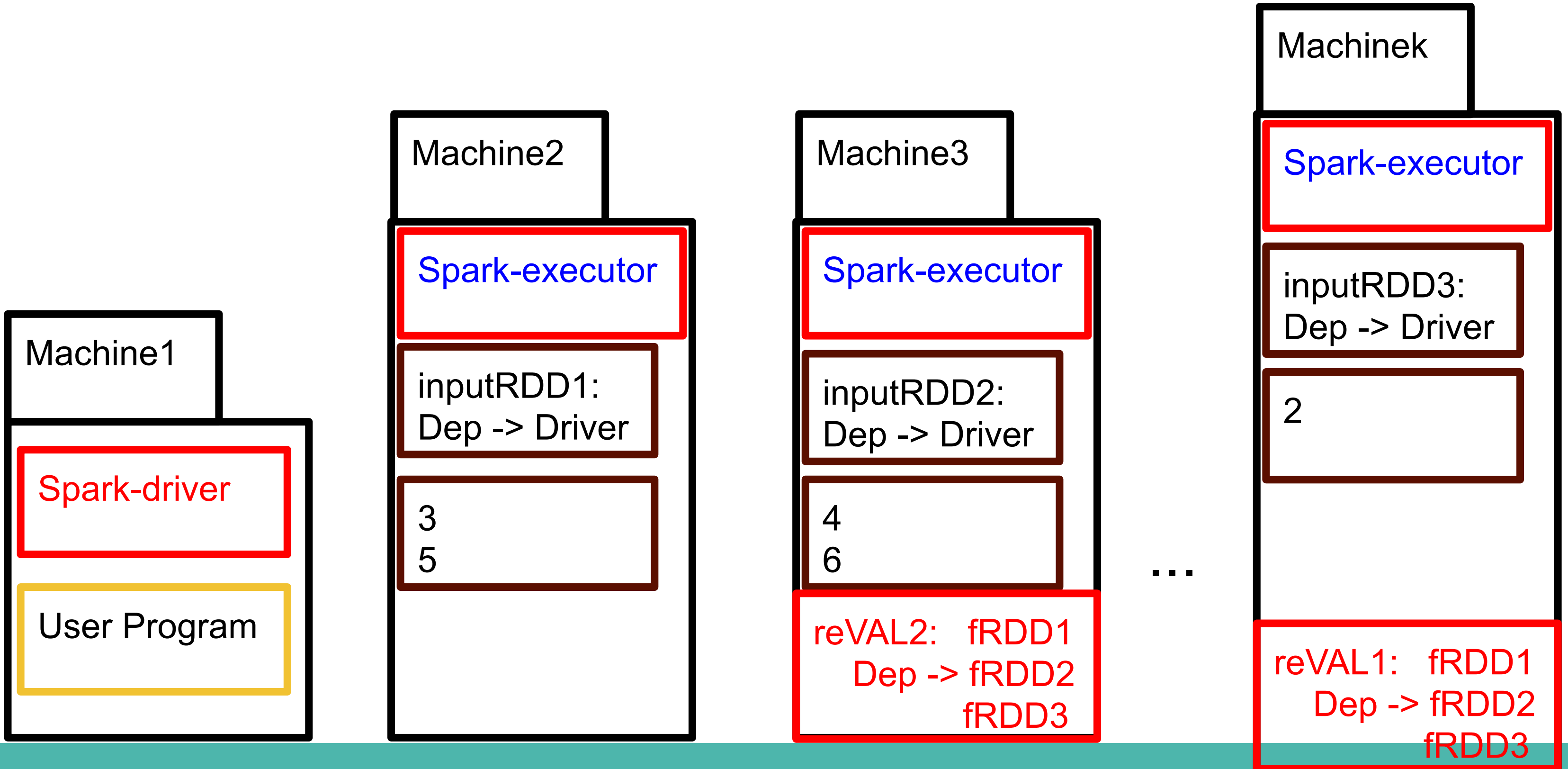


Lineage: Lazy Evaluation and Persistance



Executing the second action take turns into computing:

```
inputRDD  = sc.parallelize( [ 1, 2, 3, 4, 5 ] )
mappedRDD  = inputRDD.map( lambda elem : elem + 1 )
resVAL1 = mappedRDD.count()
resVAL2 = mappedRDD.take(2)
```



Machine1

Spark-driver

User Program

Machine2

Spark-executor

inputRDD1:
Dep -> Driver

3
5

Machine3

Spark-executor

inputRDD2:
Dep -> Driver

4
6

reVAL2: fRDD1
Dep -> fRDD2
fRDD3

…

Machinek

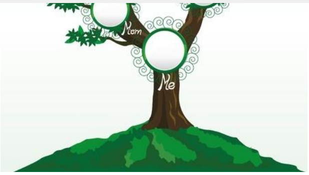
Spark-executor

inputRDD3:
Dep -> Driver

2

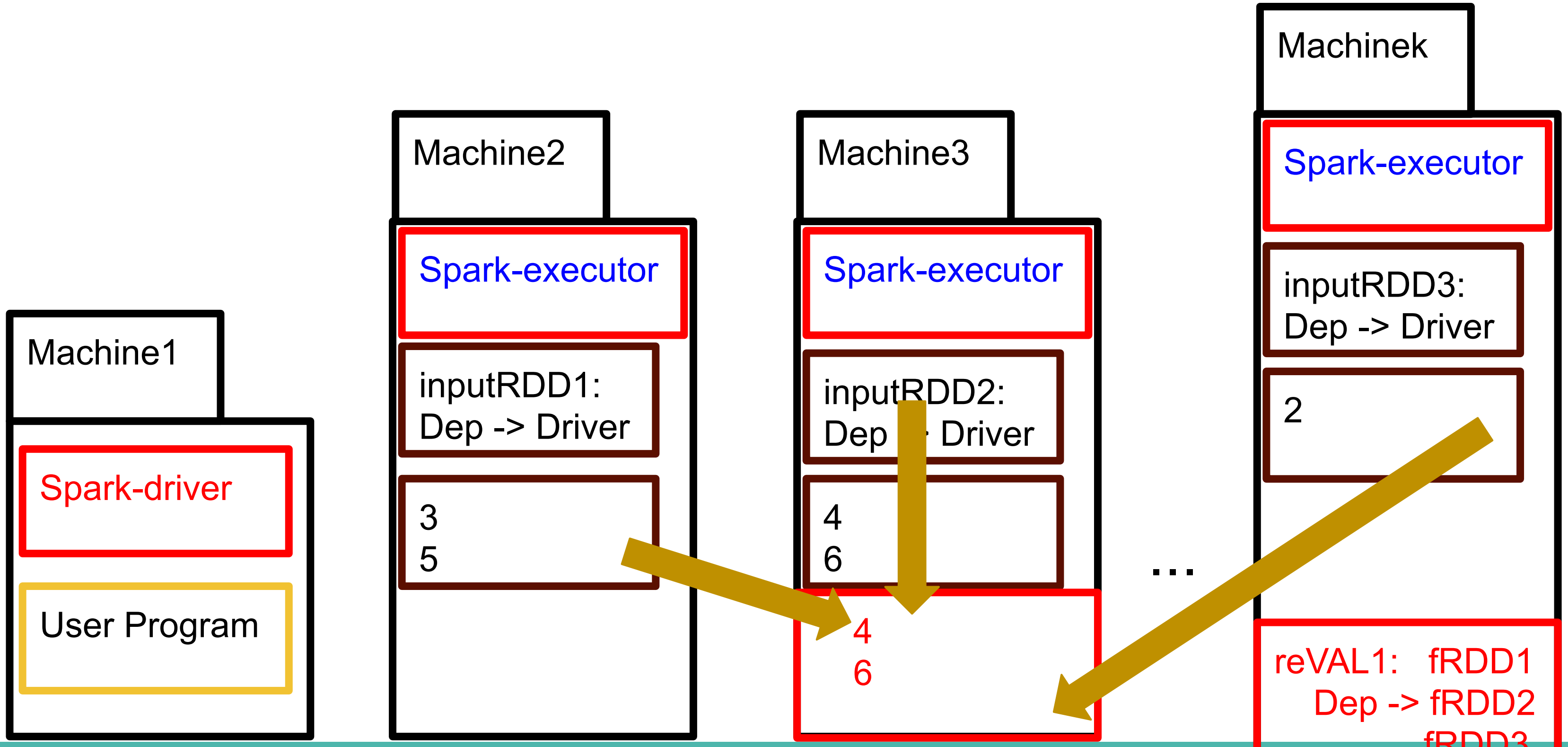
reVAL1: fRDD1
Dep -> fRDD2
fRDD3

Lineage: Lazy Evaluation and Persistance

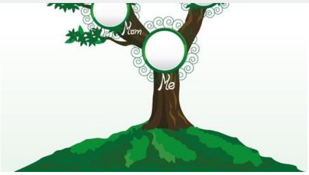


Executing the second action take turns into computing:

```
inputRDD  = sc.parallelize( [ 1, 2, 3, 4, 5 ] )
mappedRDD  = inputRDD.map( lambda elem : elem + 1 )
resVAL1 = mappedRDD.count()
resVAL2 = mappedRDD.take(2)
```

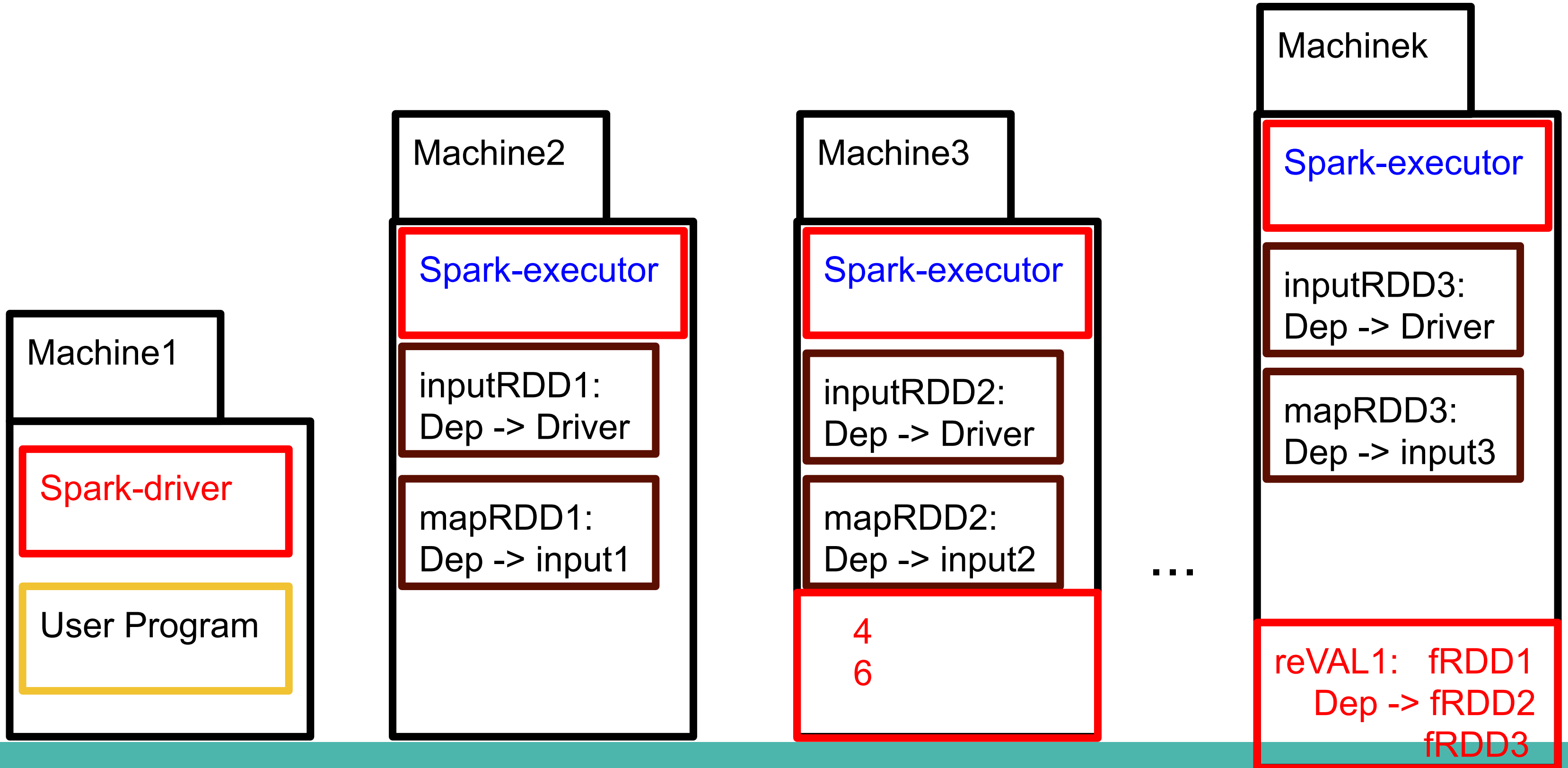


Lineage: Lazy Evaluation and Persistance

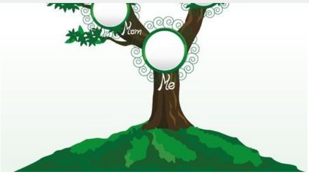


Executing the second action take turns into computing:

```
inputRDD  = sc.parallelize( [ 1, 2, 3, 4, 5 ] )
mappedRDD  = inputRDD.map( lambda elem : elem + 1 )
resVAL1 = mappedRDD.count()
resVAL2 = mappedRDD.take(2)
```

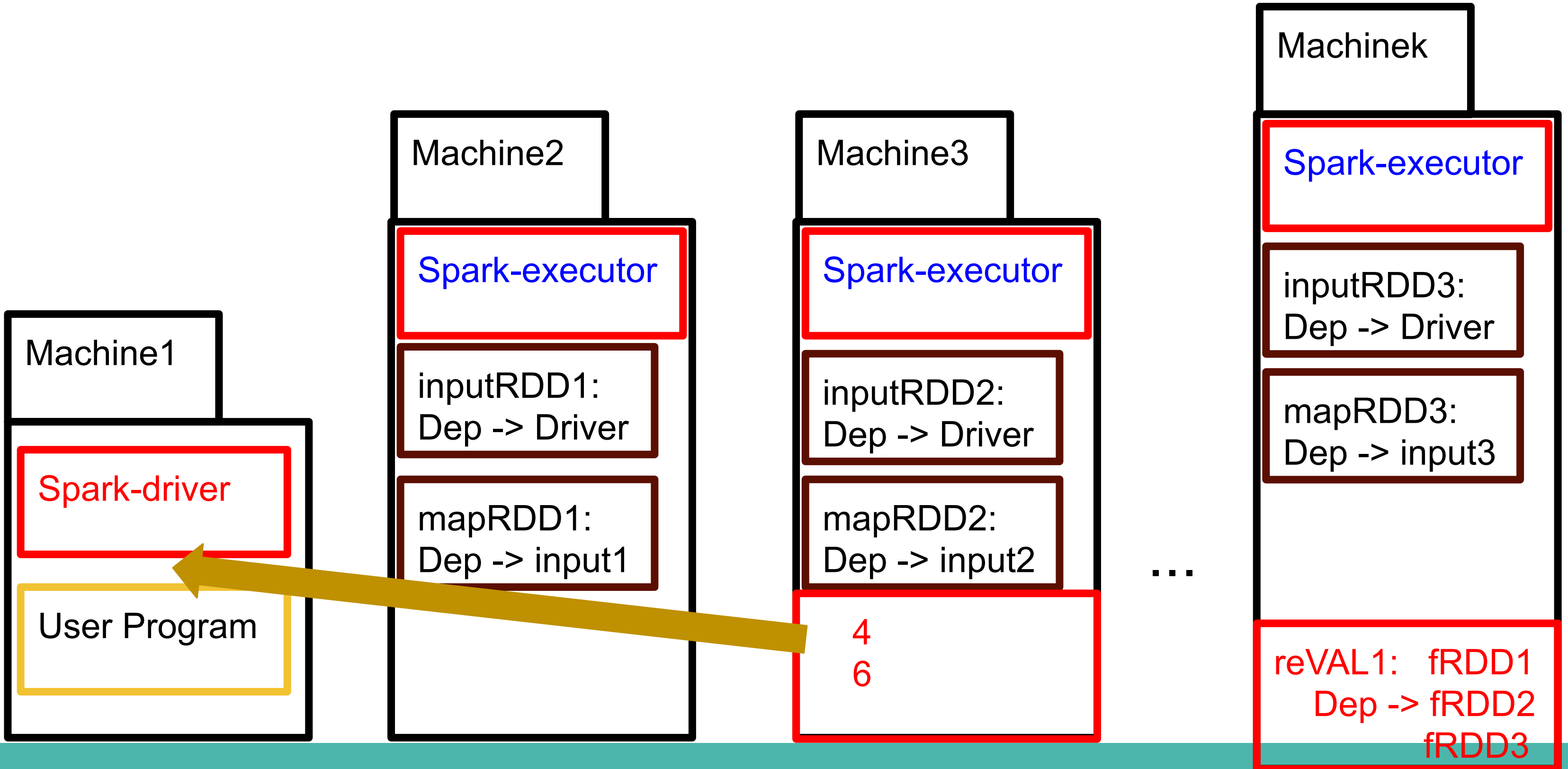


Lineage: Lazy Evaluation and Persistance

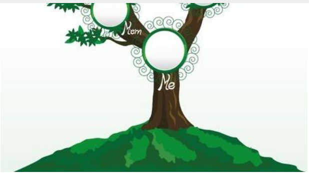


Executing the second action take turns into computing:

```
inputRDD  = sc.parallelize( [ 1, 2, 3, 4, 5 ] )
mappedRDD  = inputRDD.map( lambda elem : elem + 1 )
resVAL1 = mappedRDD.count()
resVAL2 = mappedRDD.take(2)
```

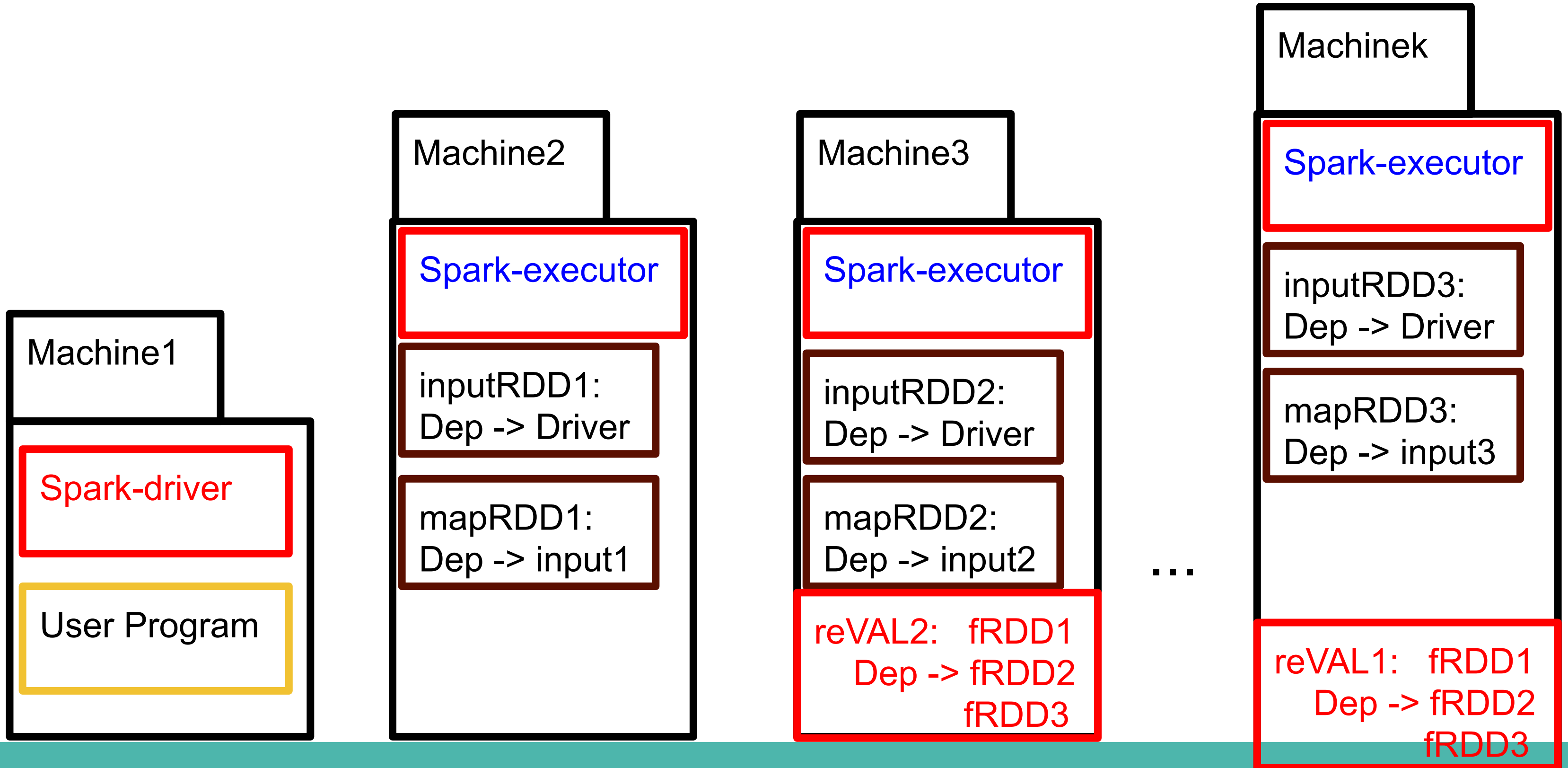


Lineage: Lazy Evaluation and Persistance



And the program finishes

```
inputRDD  = sc.parallelize( [ 1, 2, 3, 4, 5 ] )
mappedRDD  = inputRDD.map( lambda elem : elem + 1 )
resVAL1 = mappedRDD.count()
resVAL2 = mappedRDD.take(2)
```



Lineage: Lazy Evaluation and Persistance

- The motivation for removing the RDD partitions as soon as they have been used is pretty simple:
  - We are in a Big Data environment!
    Resources are scarce, so we want to keep the memory of our **Spark Executor Processes** as free as possible!

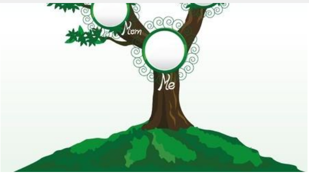
- While this idea looks wonderful on itself, it has a dark side:
  - What happens when an RDD partition is actually used twice?
  - Do you see the problem? Both inputRDD and mappedRDD have been computed twice. And there is no need for this.

Lineage: Lazy Evaluation and Persistance

- The motivation for removing the RDD partitions as soon as they have been used is pretty simple:
  - We are in a Big Data environment!
    Resources are scarce, so we want to keep the memory of our **Spark Executor Processes** as free as possible!

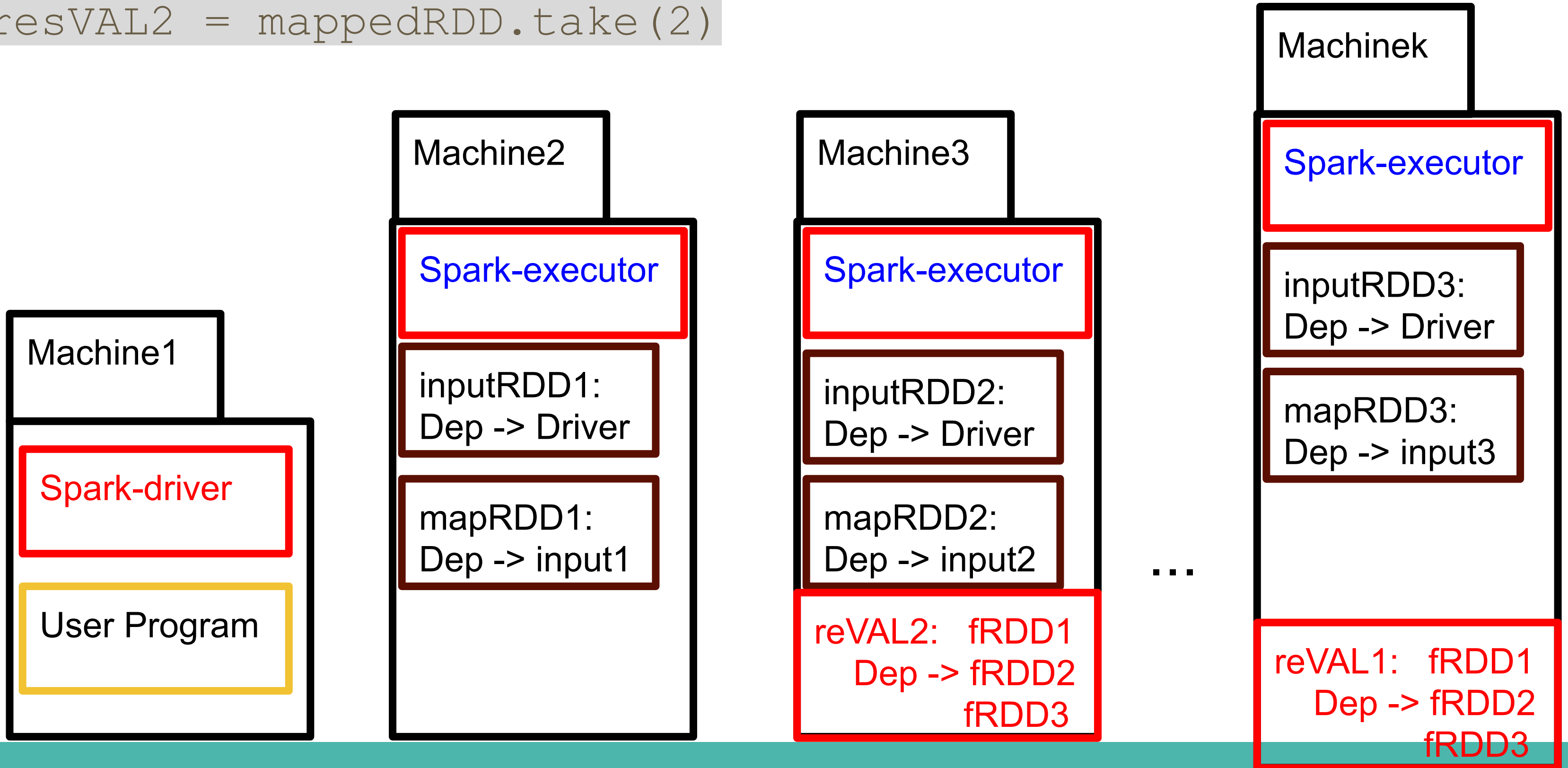
- While this idea looks wonderful on itself, it has a dark side:
  - What happens when an RDD partition is actually used twice?
  - Do you see the problem? Both inputRDD and mappedRDD have been computed twice. And there is no need for this.
  - To avoid this undesirable situation, as we mentioned in last lecture, we just need to persist each RDD being used more than once.
  - This will make that the RDD is computed, and actually kept in memory afterwards until the end of the program.

Lineage: Lazy Evaluation and Persistance



See the following program once fixed

```
inputRDD  = sc.parallelize( [ 1, 2, 3, 4, 5 ] )
mappedRDD  = inputRDD.map( lambda elem : elem + 1 )
mappedRDD.persist()
resVAL1 = mappedRDD.count()
resVAL2 = mappedRDD.take(2)
```



Machine1
Spark-driver
User Program

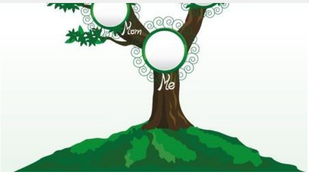
Machine2
Spark-executor
inputRDD1: Dep -> Driver
mapRDD1: Dep -> input1

Machine3
Spark-executor
inputRDD2: Dep -> Driver
mapRDD2: Dep -> input2
reVAL2: fRDD1 Dep -> fRDD2 fRDD3

...

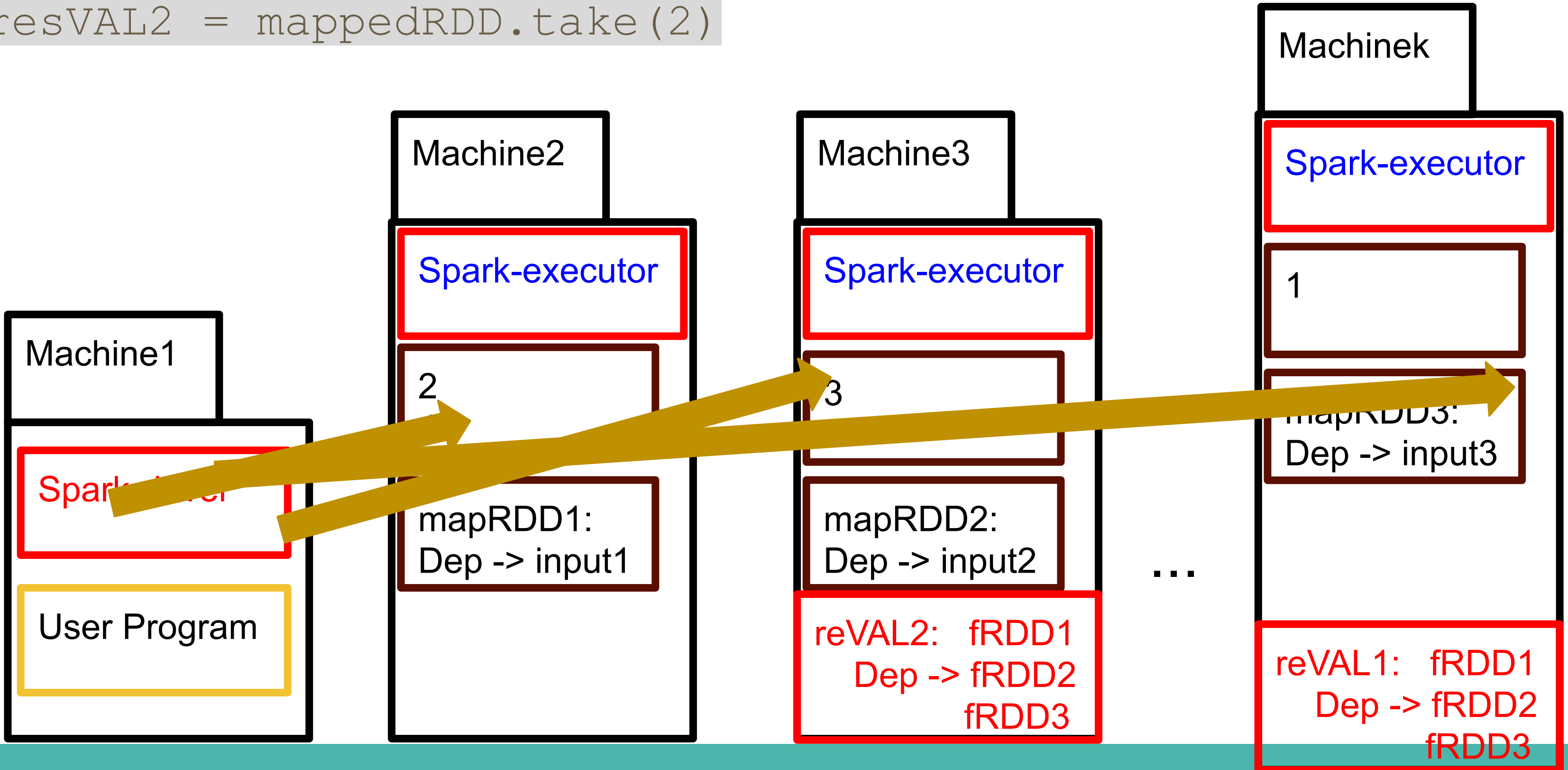
Machinek
Spark-executor
inputRDD3: Dep -> Driver
mapRDD3: Dep -> input3
reVAL1: fRDD1 Dep -> fRDD2 fRDD3

Lineage: Lazy Evaluation and Persistance



Executing the first action count turns into computing:

```
inputRDD  = sc.parallelize( [ 1, 2, 3, 4, 5 ] )
mappedRDD  = inputRDD.map( lambda elem : elem + 1 )
mappedRDD.persist()
resVAL1 = mappedRDD.count()
resVAL2 = mappedRDD.take(2)
```



Machine1

Spark-driver

User Program

Machine2

Spark-executor

2

mapRDD1:
Dep -> input1

Machine3

Spark-executor

3

mapRDD2:
Dep -> input2

reVAL2: fRDD1
Dep -> fRDD2
fRDD3

…

Machinek

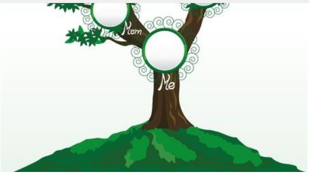
Spark-executor

1

mapRDD3:
Dep -> input3

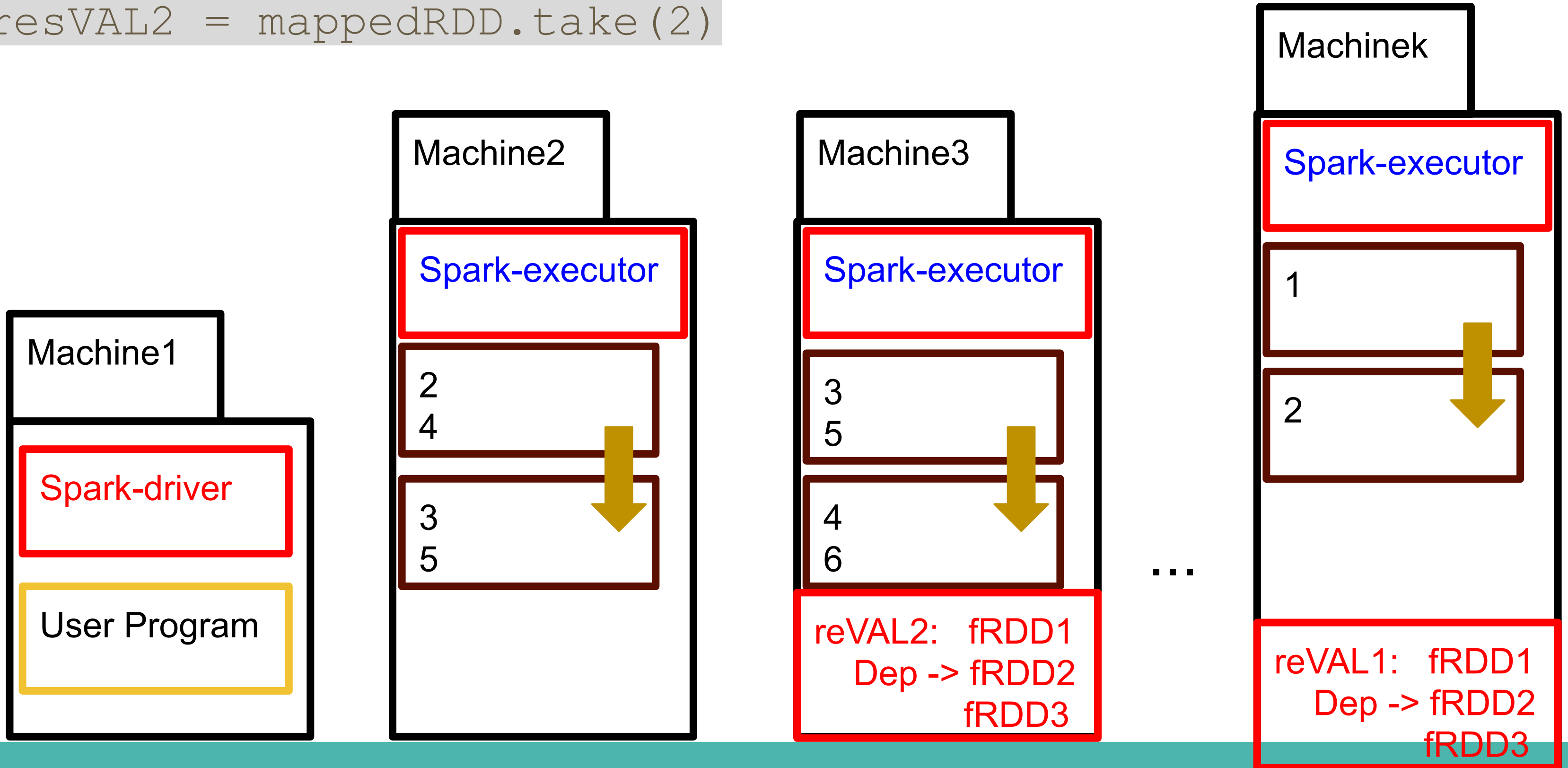
reVAL1: fRDD1
Dep -> fRDD2
fRDD3

Lineage: Lazy Evaluation and Persistance



Executing the first action count turns into computing:

```
inputRDD  = sc.parallelize( [ 1, 2, 3, 4, 5 ] )
mappedRDD  = inputRDD.map( lambda elem : elem + 1 )
mappedRDD.persist()
resVAL1 = mappedRDD.count()
resVAL2 = mappedRDD.take(2)
```



Machine1

Spark-driver

User Program

Machine2

Spark-executor

2
4

3
5

Machine3

Spark-executor

3
5

4
6

reVAL2: fRDD1
Dep -> fRDD2
fRDD3

…

Machinek

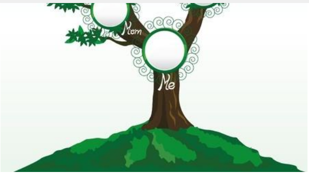
Spark-executor

1

2

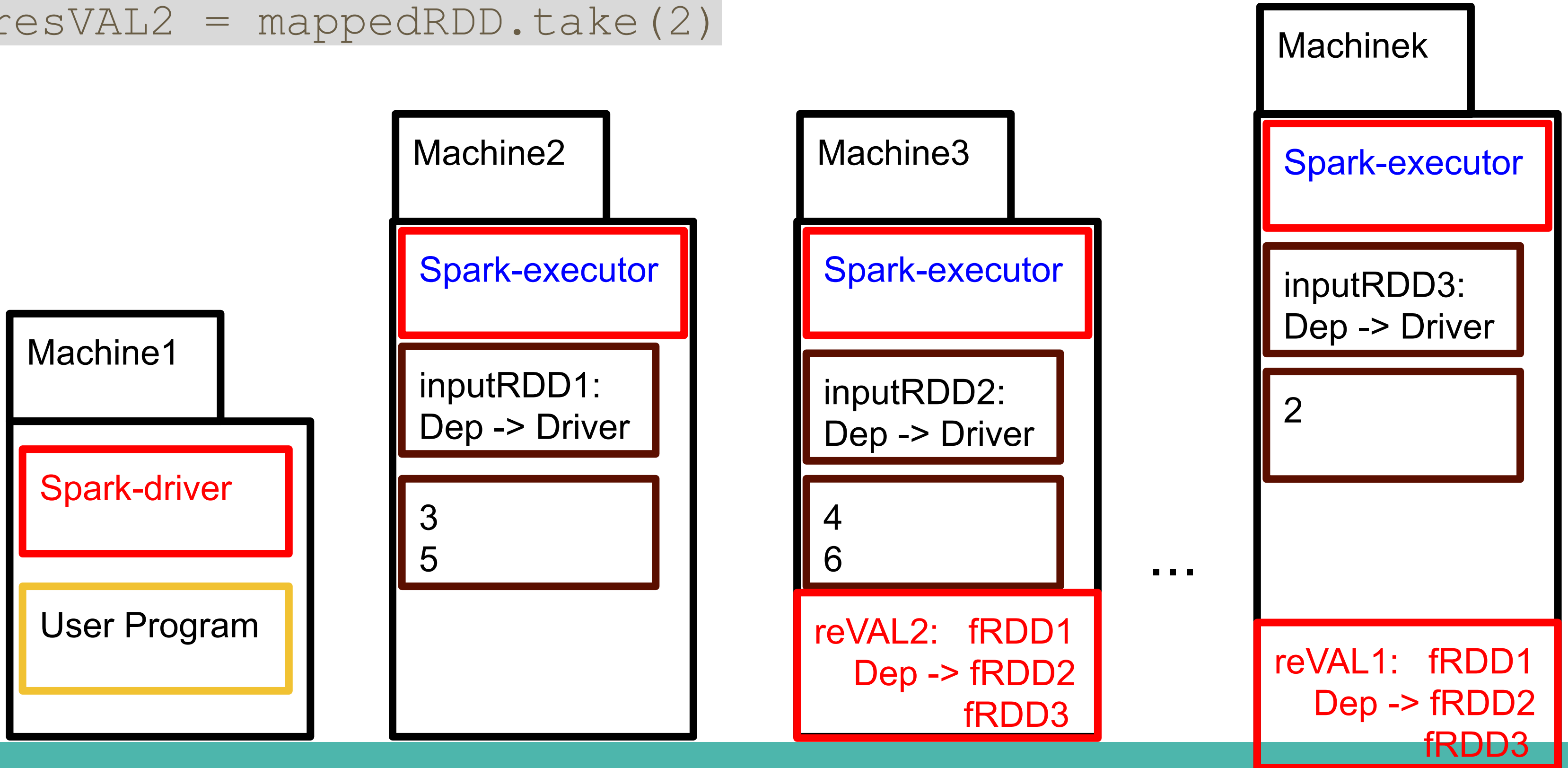
reVAL1: fRDD1
Dep -> fRDD2
fRDD3

Lineage: Lazy Evaluation and Persistance

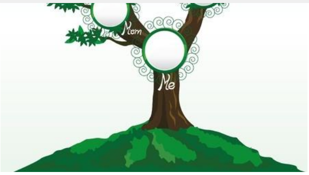


Executing the first action count turns into computing:

```
inputRDD   = sc.parallelize( [ 1, 2, 3, 4, 5 ] )
mappedRDD  = inputRDD.map( lambda elem : elem + 1 )
mappedRDD.persist()
resVAL1 = mappedRDD.count()
resVAL2 = mappedRDD.take(2)
```

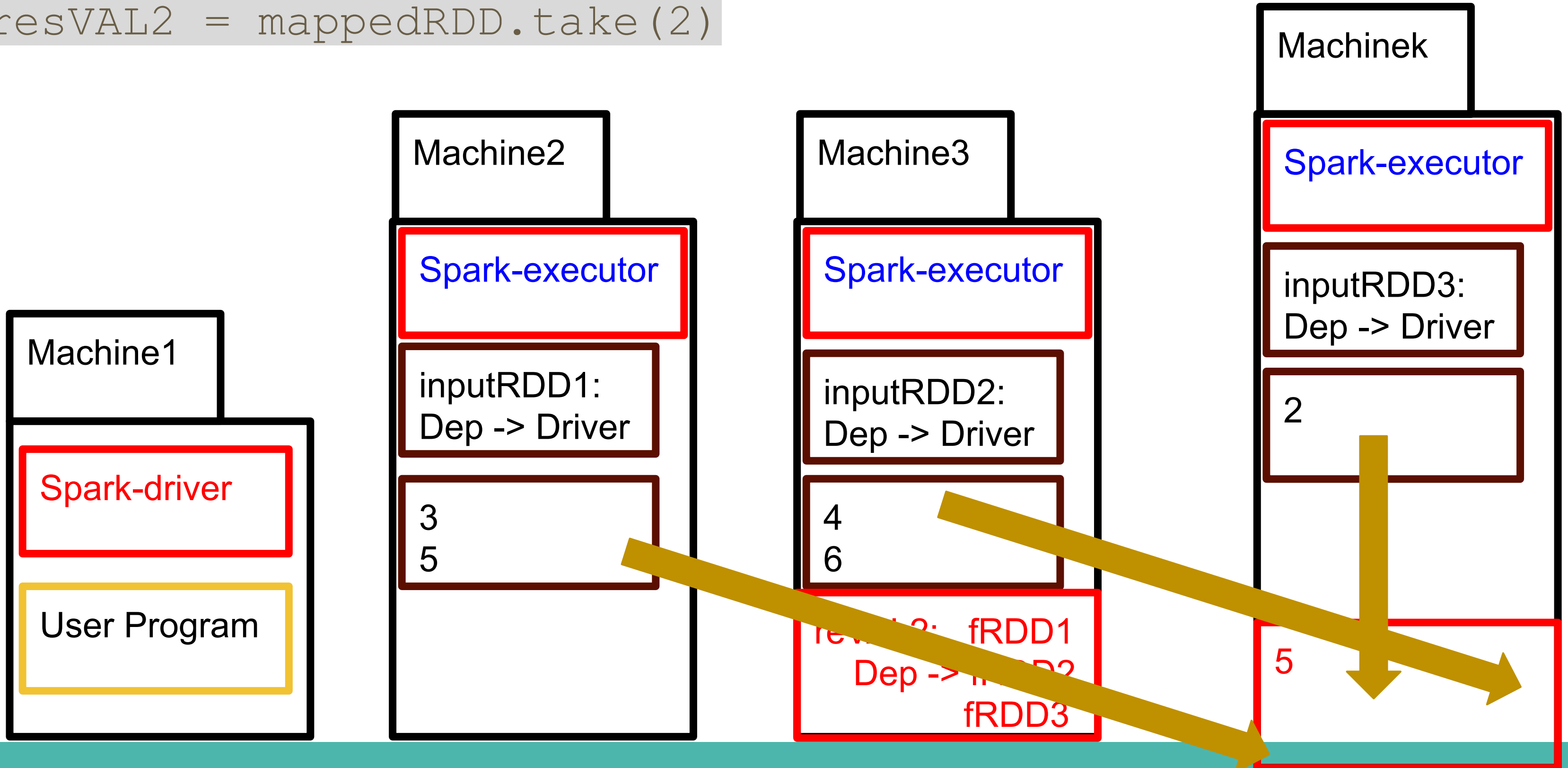


Lineage: Lazy Evaluation and Persistance

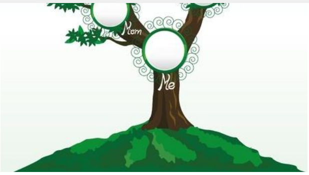


Executing the first action count turns into computing:

```
inputRDD  = sc.parallelize( [ 1, 2, 3, 4, 5 ] )
mappedRDD  = inputRDD.map( lambda elem : elem + 1 )
mappedRDD.persist()
resVAL1 = mappedRDD.count()
resVAL2 = mappedRDD.take(2)
```



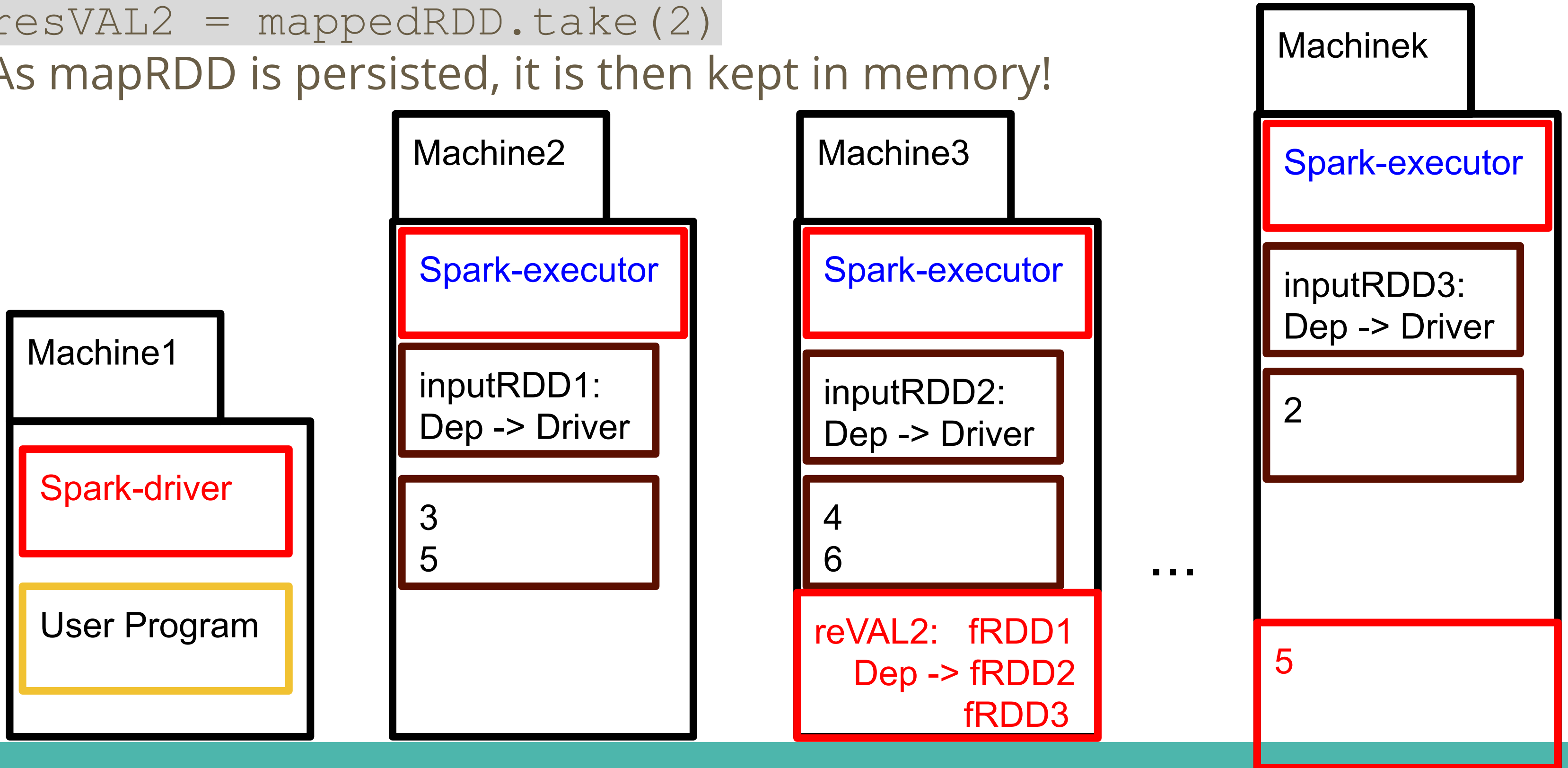
Lineage: Lazy Evaluation and Persistance



Executing the first action count turns into computing:

```
inputRDD  = sc.parallelize( [ 1, 2, 3, 4, 5 ] )
mappedRDD  = inputRDD.map( lambda elem : elem + 1 )
mappedRDD.persist()
resVAL1 = mappedRDD.count()
resVAL2 = mappedRDD.take(2)
```

As mapRDD is persisted, it is then kept in memory!



Machine1

Spark-driver

User Program

Machine2

Spark-executor

inputRDD1: Dep -> Driver

3
5

Machine3

Spark-executor

inputRDD2: Dep -> Driver

4
6

reVAL2: fRDD1
Dep -> fRDD2
fRDD3

…

Machinek

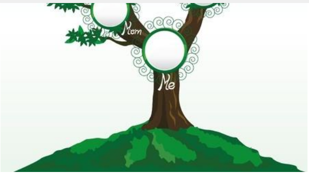
Spark-executor

inputRDD3: Dep -> Driver

2

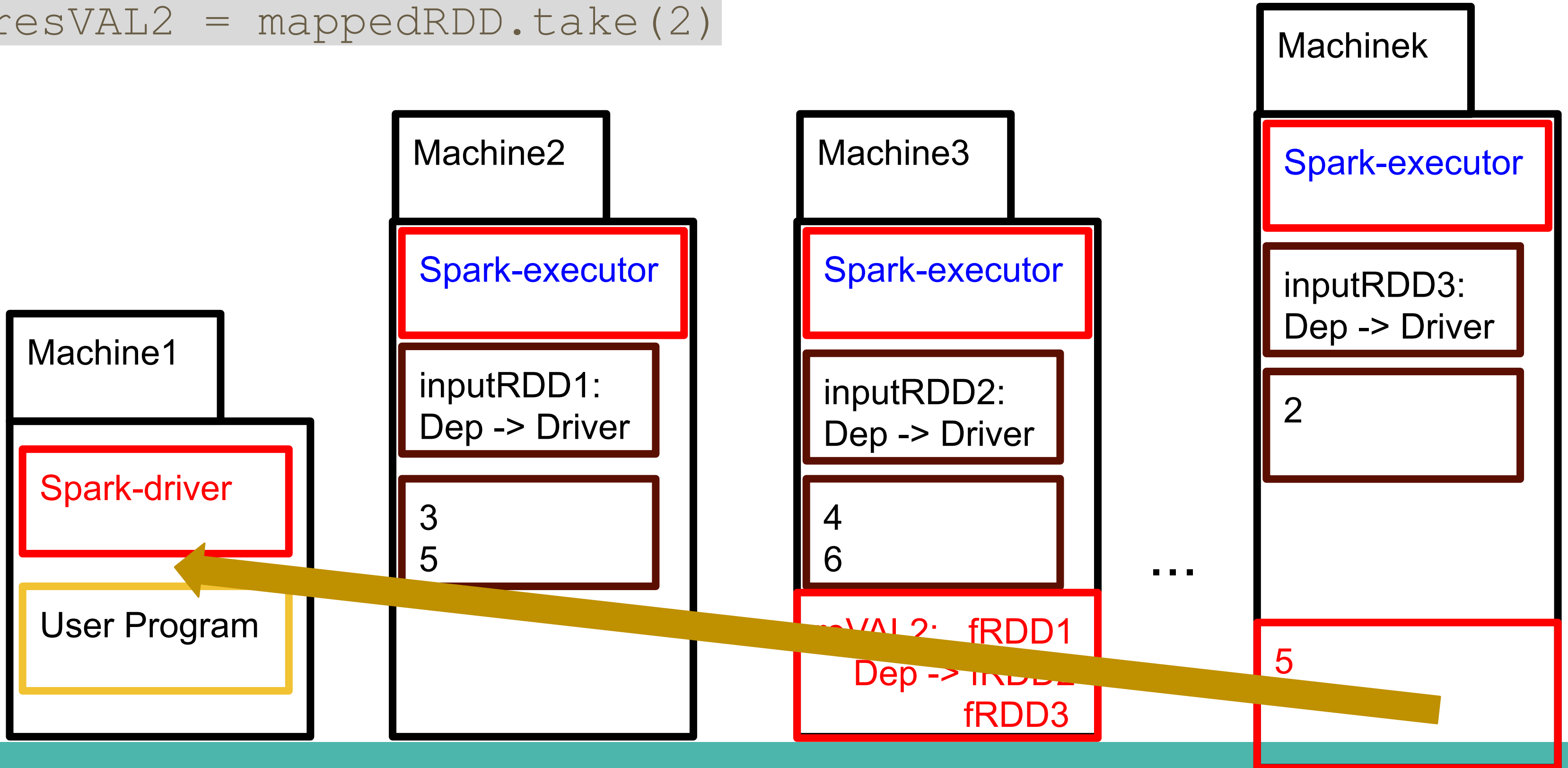
5

Lineage: Lazy Evaluation and Persistance

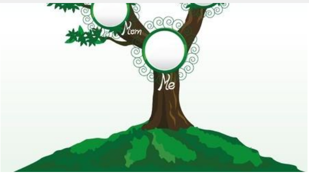


Executing the first action count turns into computing:

```
inputRDD  = sc.parallelize( [ 1, 2, 3, 4, 5 ] )
mappedRDD  = inputRDD.map( lambda elem : elem + 1 )
mappedRDD.persist()
resVAL1 = mappedRDD.count()
resVAL2 = mappedRDD.take(2)
```

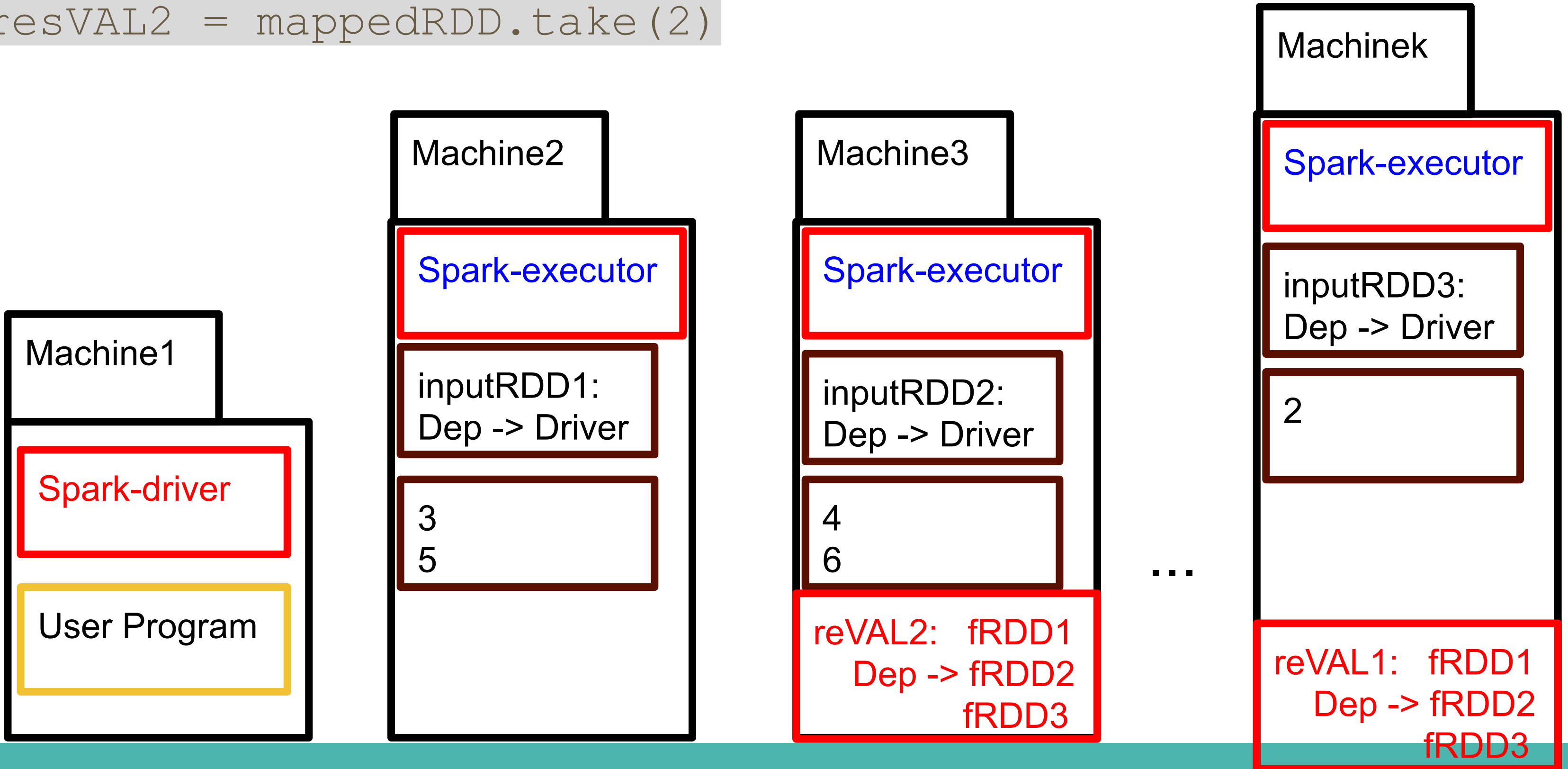


Lineage: Lazy Evaluation and Persistance

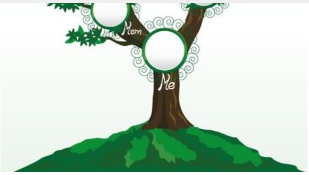


Executing the first action count turns into computing:

```
inputRDD  = sc.parallelize( [ 1, 2, 3, 4, 5 ] )
mappedRDD  = inputRDD.map( lambda elem : elem + 1 )
mappedRDD.persist()
resVAL1 = mappedRDD.count()
resVAL2 = mappedRDD.take(2)
```



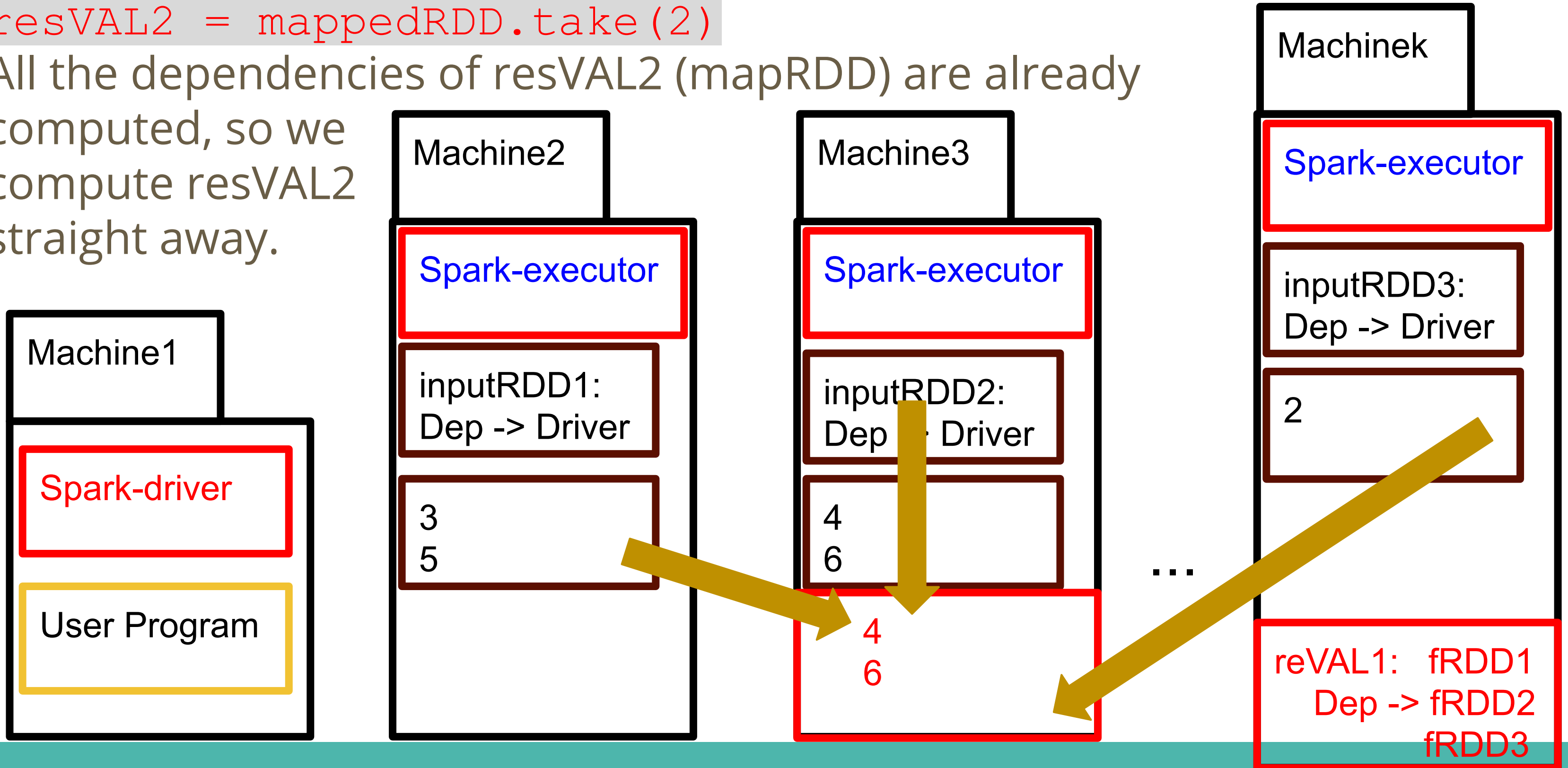
Lineage: Lazy Evaluation and Persistance



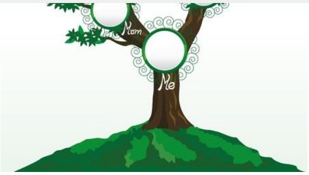
Executing the second action take becomes trivial now:

```
inputRDD  = sc.parallelize( [ 1, 2, 3, 4, 5 ] )
mappedRDD  = inputRDD.map( lambda elem : elem + 1 )
mappedRDD.persist()
resVAL1 = mappedRDD.count()
resVAL2 = mappedRDD.take(2)
```

All the dependencies of resVAL2 (mapRDD) are already computed, so we compute resVAL2 straight away.



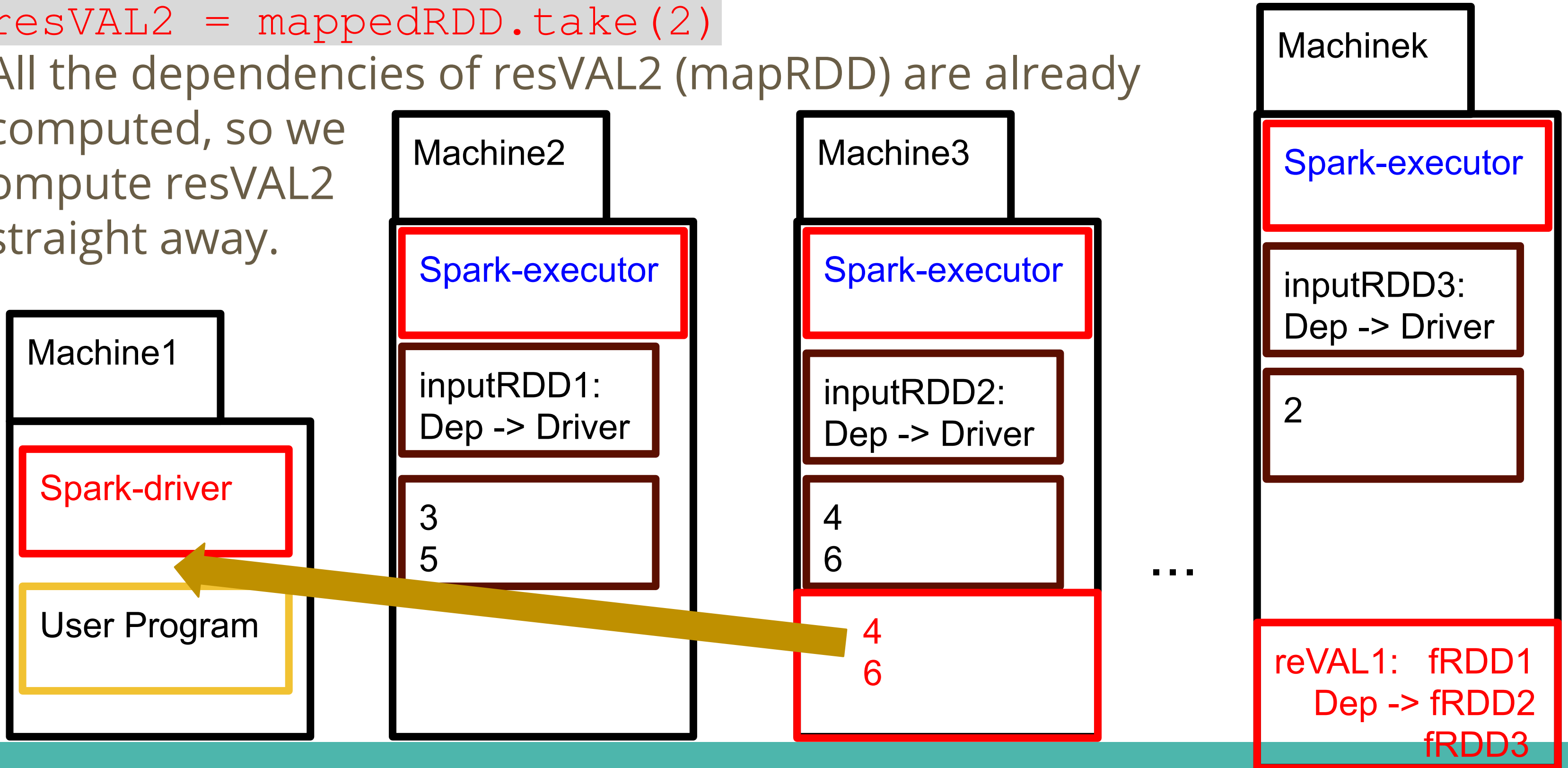
Lineage: Lazy Evaluation and Persistance



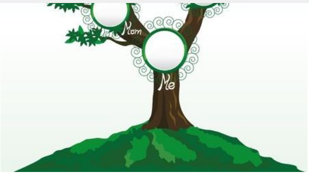
Executing the second action take becomes trivial now:

```
inputRDD  = sc.parallelize( [ 1, 2, 3, 4, 5 ] )
mappedRDD  = inputRDD.map( lambda elem : elem + 1 )
mappedRDD.persist()
resVAL1 = mappedRDD.count()
resVAL2 = mappedRDD.take(2)
```

All the dependencies of resVAL2 (mapRDD) are already computed, so we ompute resVAL2 straight away.

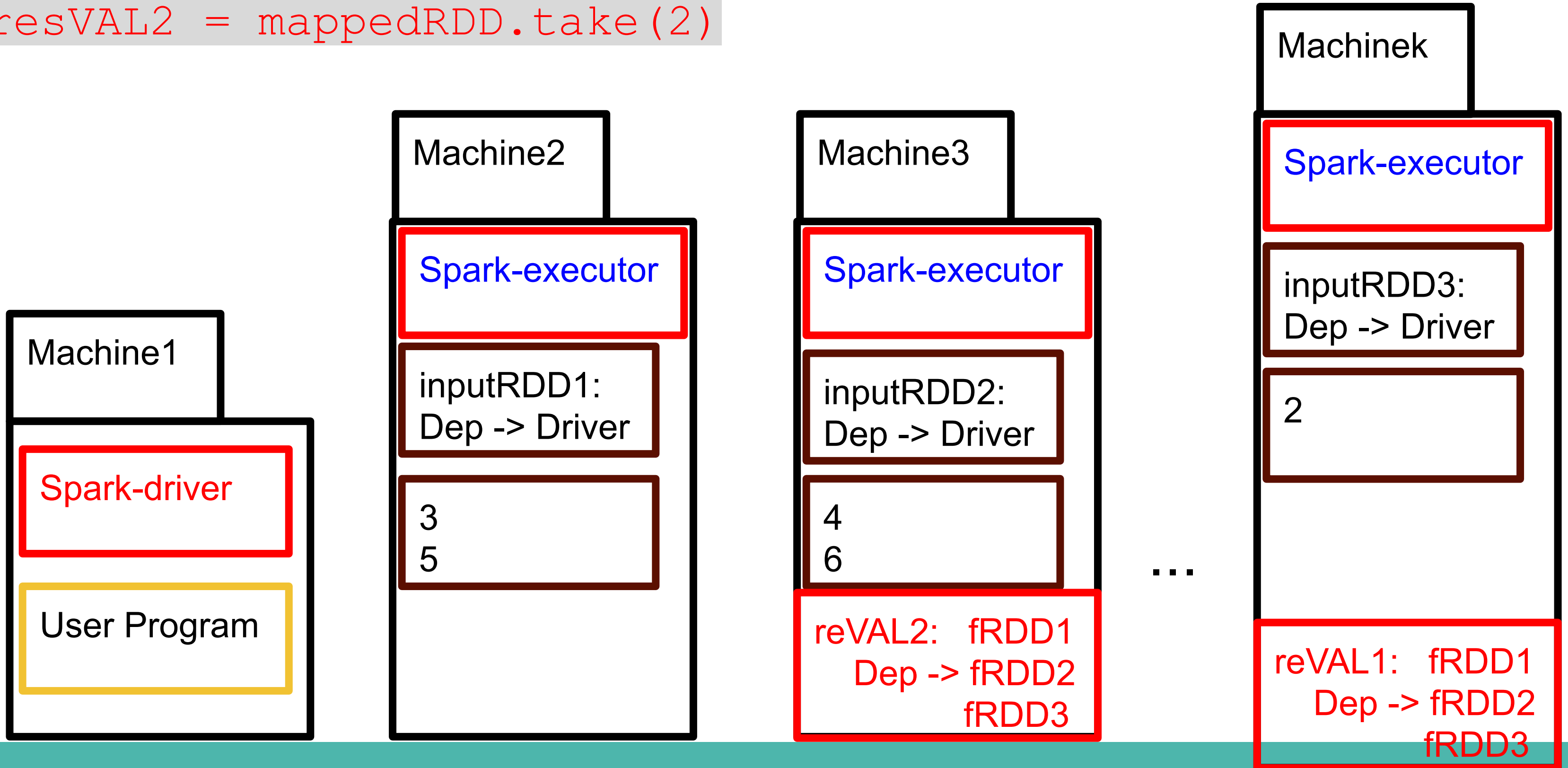


Lineage: Lazy Evaluation and Persistance



Executing the second action take turns into computing:

```
inputRDD  = sc.parallelize( [ 1, 2, 3, 4, 5 ] )
mappedRDD  = inputRDD.map( lambda elem : elem + 1 )
mappedRDD.persist()
resVAL1 = mappedRDD.count()
resVAL2 = mappedRDD.take(2)
```



Machine1

Spark-driver

User Program

Machine2

Spark-executor

inputRDD1: Dep -> Driver

3
5

Machine3

Spark-executor

inputRDD2: Dep -> Driver

4
6

reVAL2: fRDD1
Dep -> fRDD2
fRDD3

…

Machinek

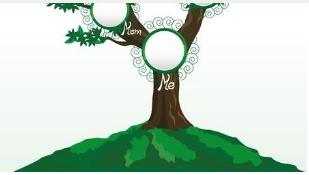
Spark-executor

inputRDD3: Dep -> Driver

2

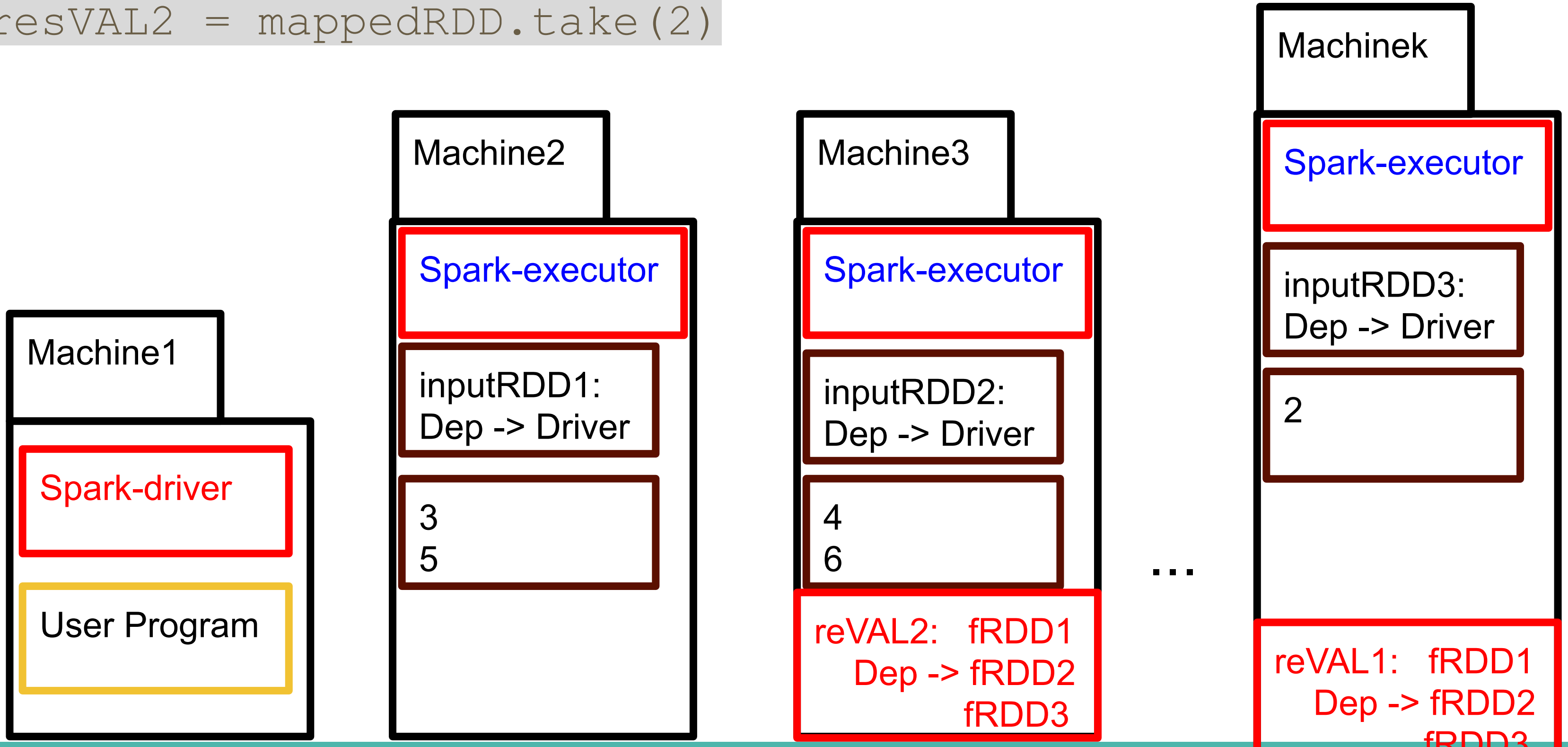
reVAL1: fRDD1
Dep -> fRDD2
fRDD3

Lineage: Lazy Evaluation and Persistance



And the program finishes

```
inputRDD  = sc.parallelize( [ 1, 2, 3, 4, 5 ] )
mappedRDD  = inputRDD.map( lambda elem : elem + 1 )
mappedRDD.persist()
resVAL1 = mappedRDD.count()
resVAL2 = mappedRDD.take(2)
```



Lineage: Lazy Evaluation and Persistance

- The motivation for removing the RDD partitions as soon as they used is pretty simple:
  - We are in a Big Data environment!
    Resources are scarce, so we want to keep the memory of our **Spark Executor Processes** as free as possible!

- While this idea looks wonderful on itself, it has a dark side:
  - What happens when an RDD partition is actually used twice?
  - Do you see the problem? Both inputRDD and mappedRDD have been computed twice. And there is no need for this.
  - To avoid this undesirable situation, as we mentioned in last lecture, we just need to persist each RDD being used more than once.
  - This will make that the RDD is computed, and actually kept in memory afterwards until the end of the program.
  - As we have seen, now our computation is efficient, and everything that is needed to be computed it is computed...just once!

Outline

1. Setting the Context.
2. RDD Private Side: Partitions and Lineage.
    a. Internal Representation.
    b. Partitions.
    c. Lineage: Narrow and Wide Transformations.
    d. Lineage: Lazy evaluation.
    e. Lineage: Lazy evaluation and Persistance.
    f. Lineage: Fault tolerant.
3. Spark Application: Jobs, Stages and Tasks.

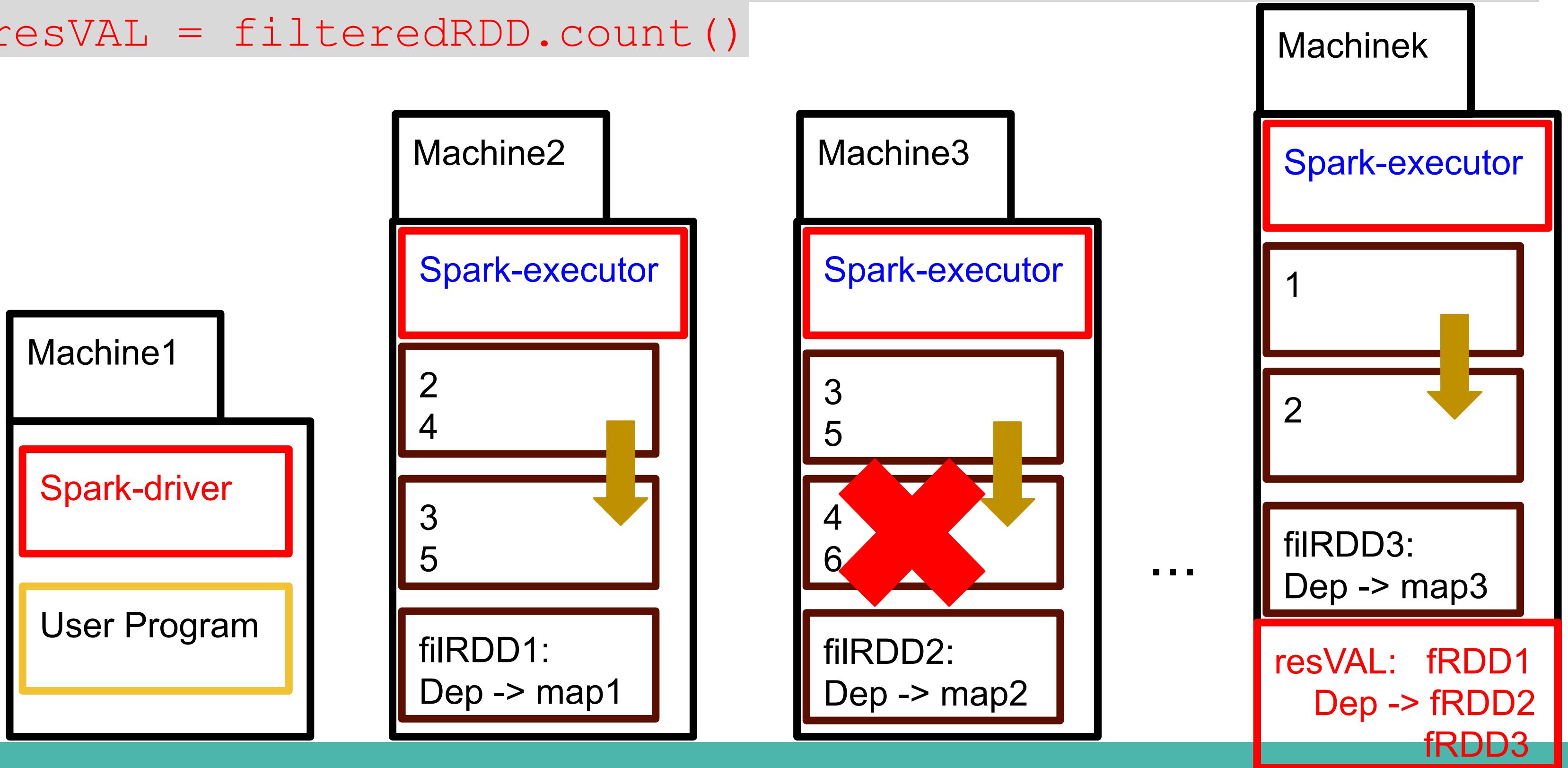
Lineage: Fault Tolerant

Now, why do we claim lineage to be crucial for becoming fault tolerant?

Lineage: Fault Tolerant

Very simple:
If, in the middle of the computation one partition gets lost…

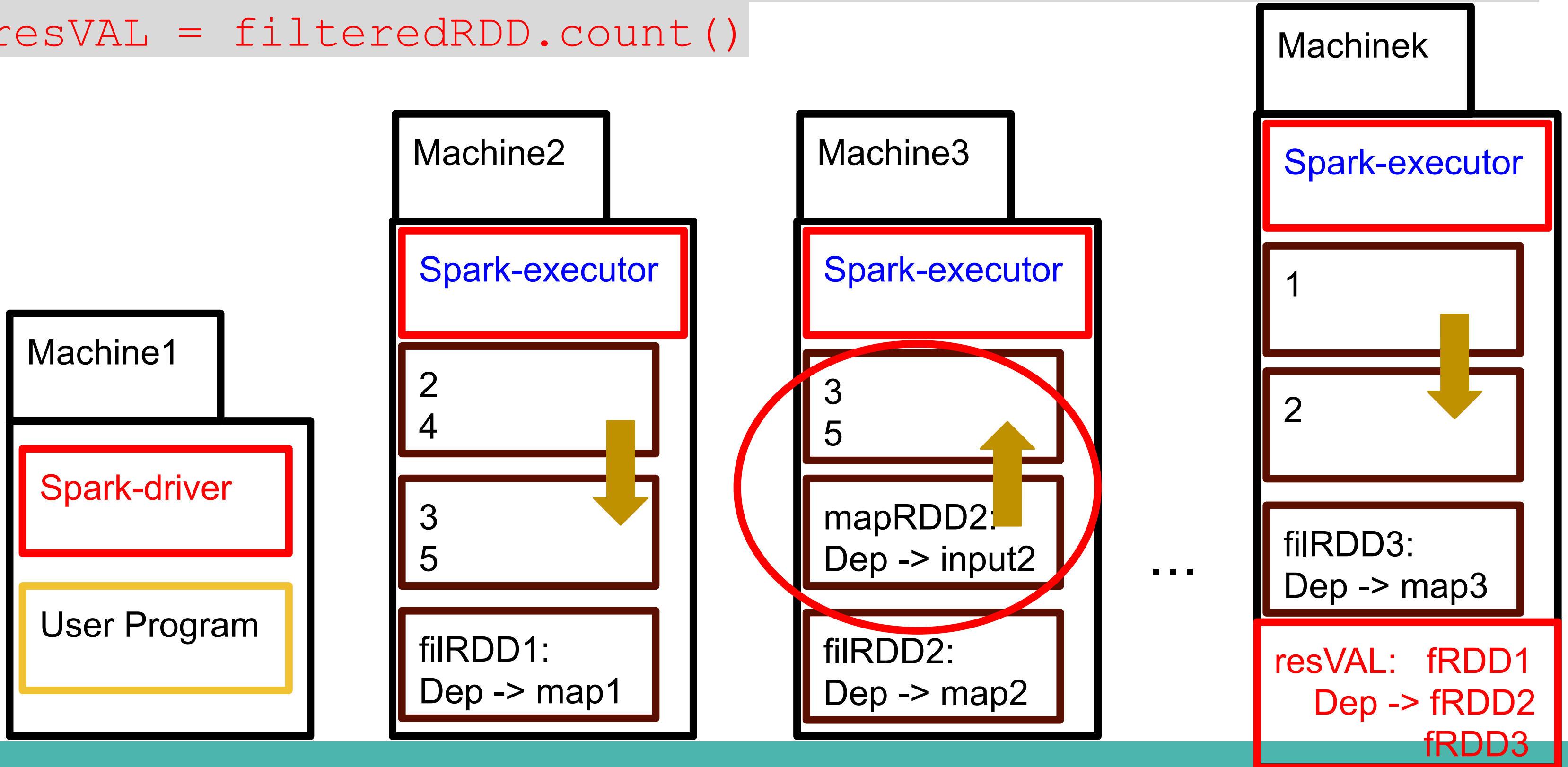
```
inputRDD  = sc.parallelize( [ 1, 2, 3, 4, 5 ] )
mappedRDD  = inputRDD.map( lambda elem : elem + 1 )
filteredRDD  = mappedRDD.filter( lambda elem : elem > 3 )
resVAL = filteredRDD.count()
```



Lineage: Fault Tolerant

Very simple:
...no worries, we use its lineage again so as to recompute it again...

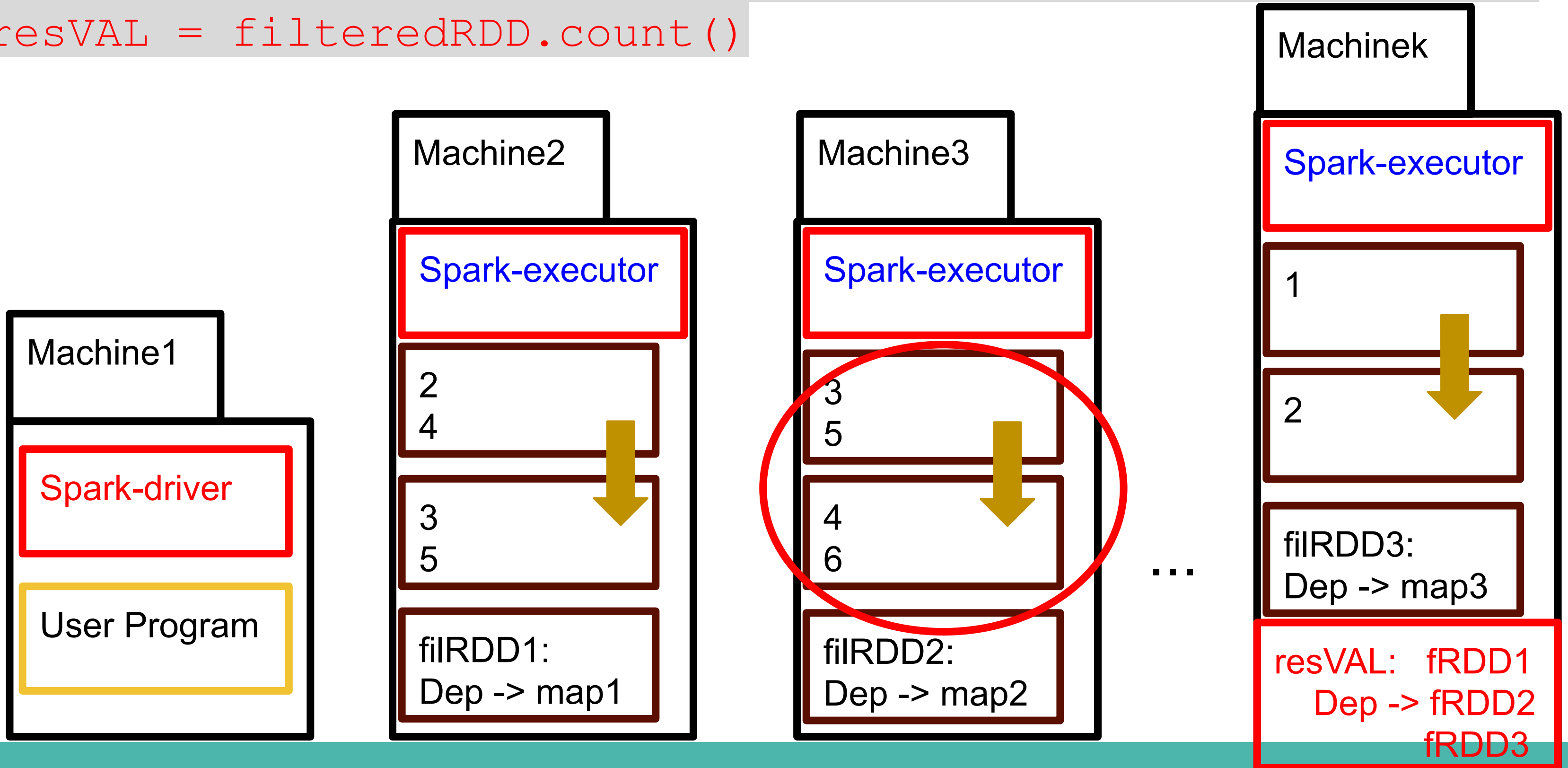
```
inputRDD  = sc.parallelize( [ 1, 2, 3, 4, 5 ] )
mappedRDD  = inputRDD.map( lambda elem : elem + 1 )
filteredRDD  = mappedRDD.filter( lambda elem : elem > 3 )
resVAL = filteredRDD.count()
```



Lineage: Fault Tolerant

Very simple:
…no worries, we use its lineage again so as to recompute it again…

```
inputRDD   = sc.parallelize( [ 1, 2, 3, 4, 5 ] )
mappedRDD  = inputRDD.map( lambda elem : elem + 1 )
filteredRDD  = mappedRDD.filter( lambda elem : elem > 3 )
resVAL = filteredRDD.count()
```



Machine1

Spark-driver

User Program

Machine2

Spark-executor

2
4

3
5

filRDD1:
Dep -> map1

Machine3

Spark-executor

3
5

4
6

filRDD2:
Dep -> map2

…

Machinek

Spark-executor

1

2

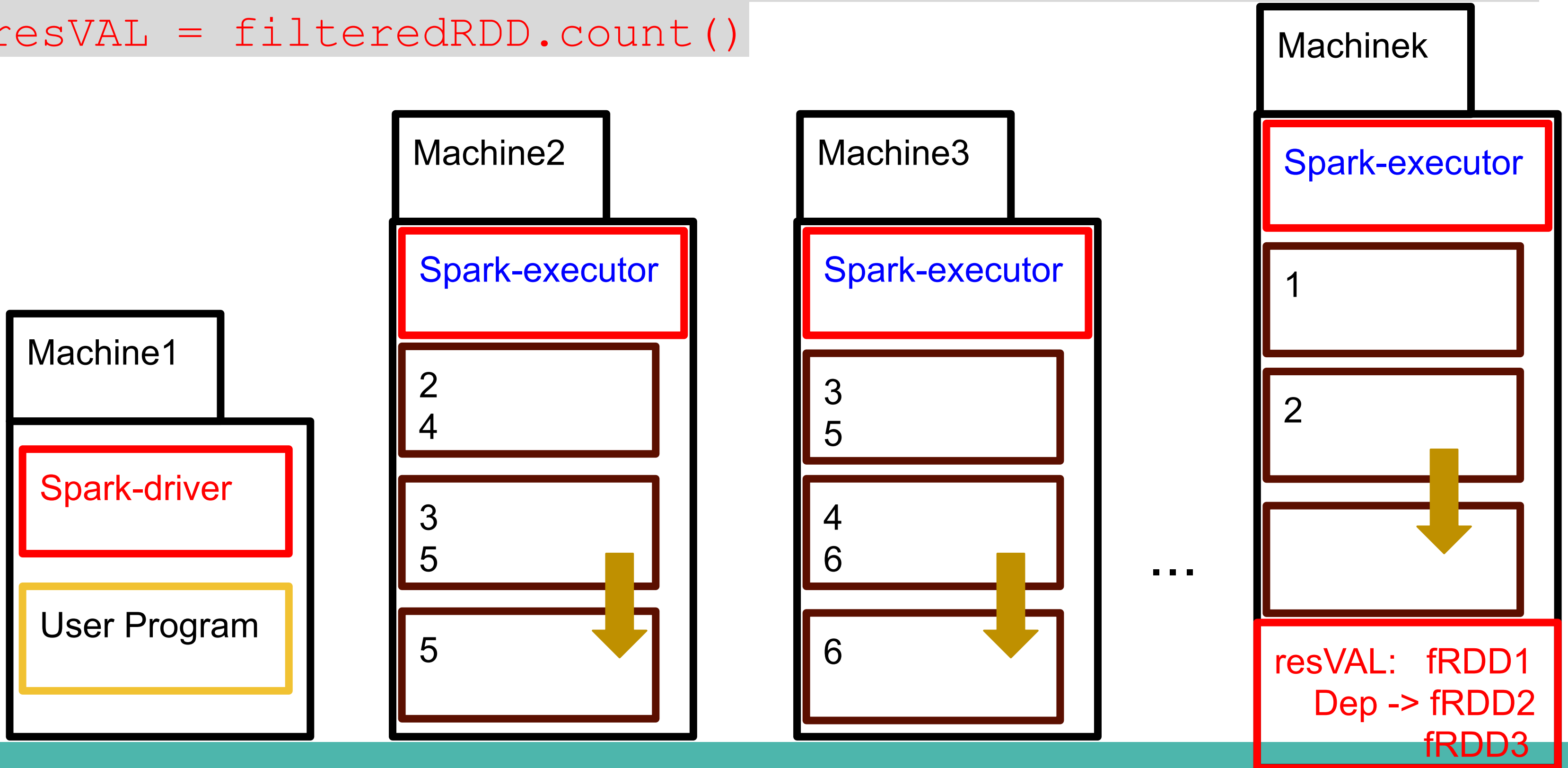
filRDD3:
Dep -> map3

resVAL: fRDD1
Dep -> fRDD2
fRDD3

Lineage: Fault Tolerant

...and we continue from there.

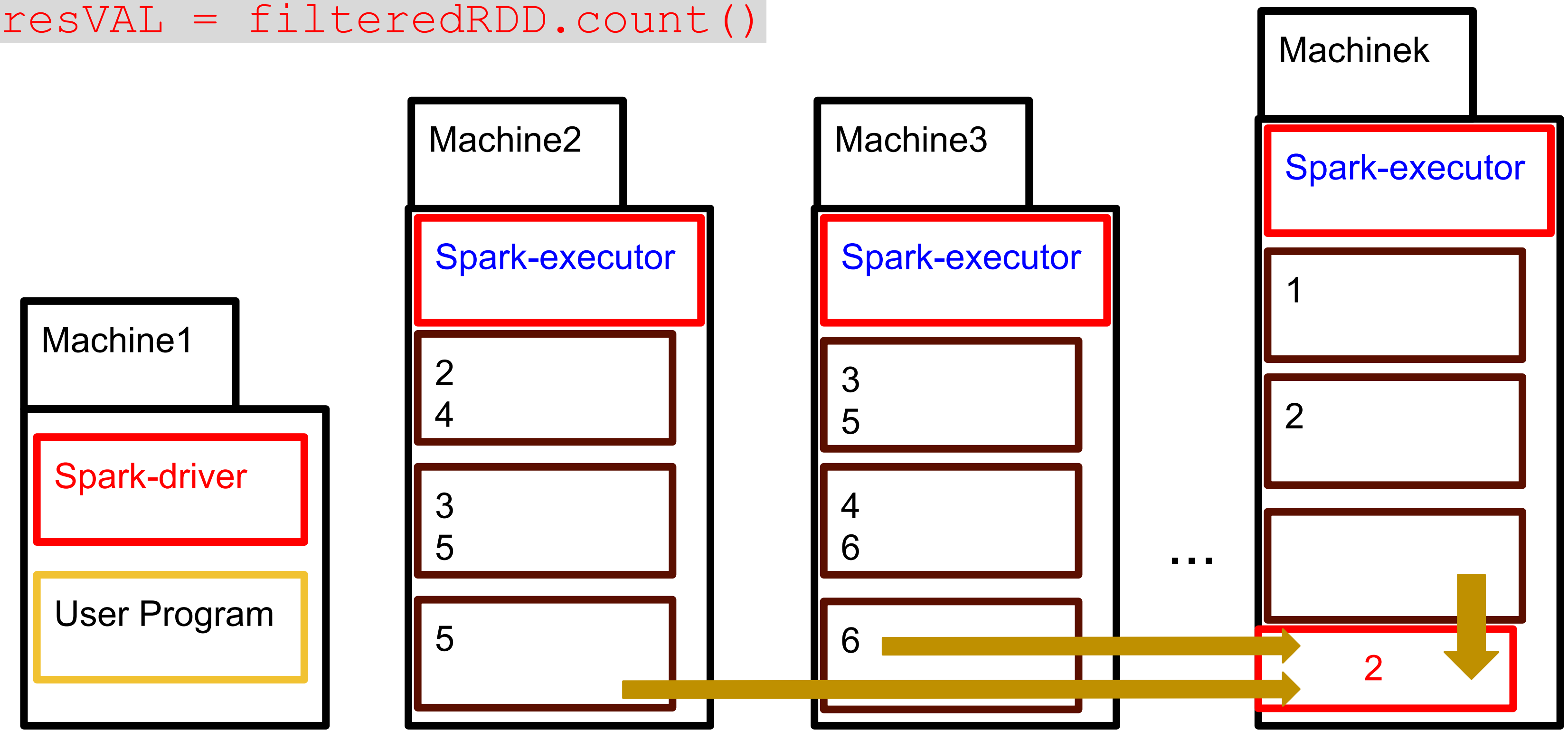
```
inputRDD   = sc.parallelize( [ 1, 2, 3, 4, 5 ] )
mappedRDD  = inputRDD.map( lambda elem : elem + 1 )
filteredRDD  = mappedRDD.filter( lambda elem : elem > 3 )
resVAL = filteredRDD.count()
```



Lineage: Fault Tolerant

...and we continue from there.

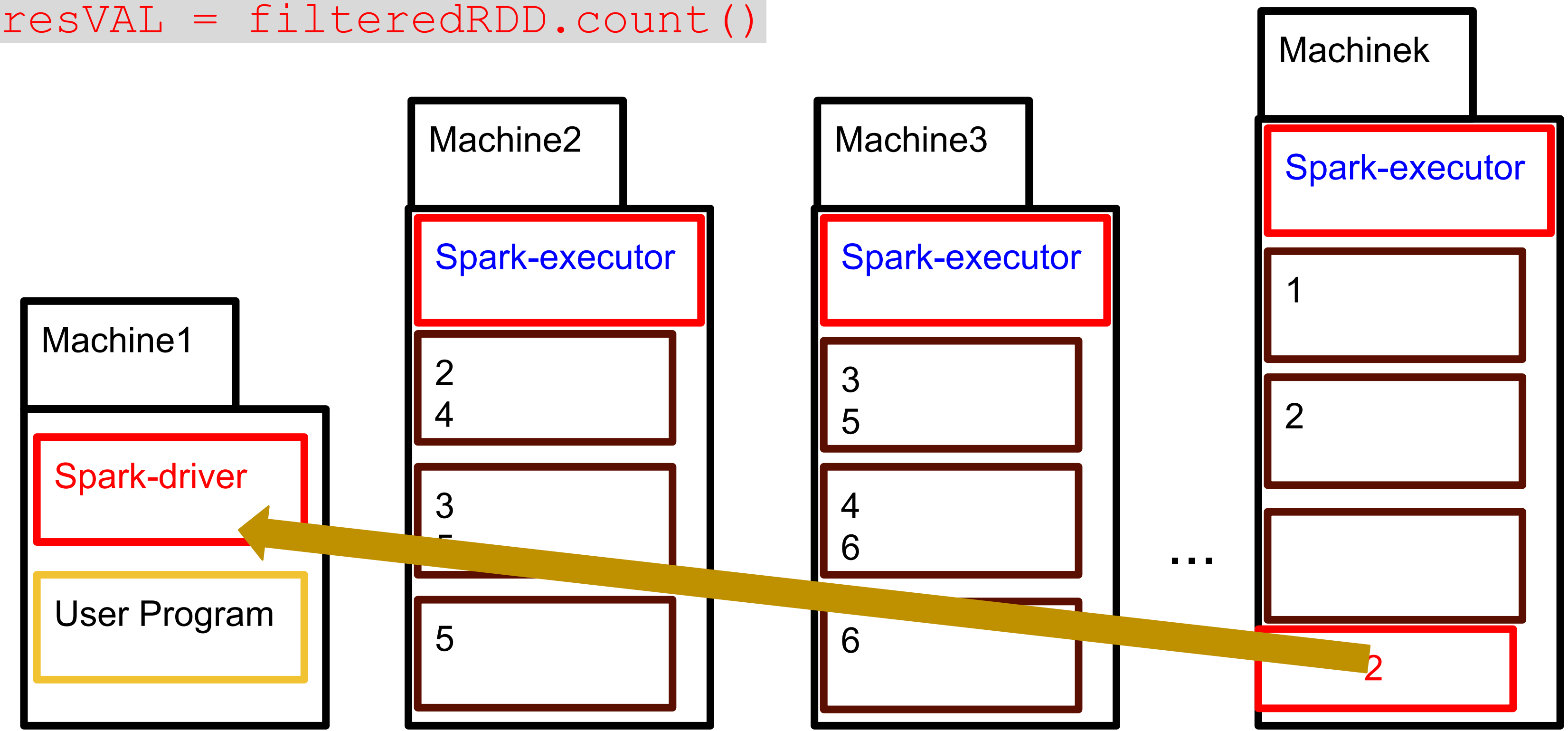
```
inputRDD  = sc.parallelize( [ 1, 2, 3, 4, 5 ] )
mappedRDD  = inputRDD.map( lambda elem : elem + 1 )
filteredRDD  = mappedRDD.filter( lambda elem : elem > 3 )
resVAL = filteredRDD.count()
```



Lineage: Fault Tolerant

...and we continue from there.

```
inputRDD   = sc.parallelize( [ 1, 2, 3, 4, 5 ] )
mappedRDD   = inputRDD.map( lambda elem : elem + 1 )
filteredRDD   = mappedRDD.filter( lambda elem : elem > 3 )
resVAL = filteredRDD.count()
```



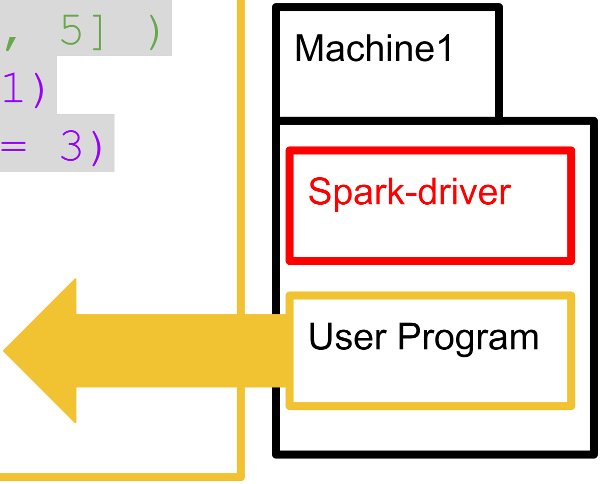
Outline

1. Setting the Context.
2. RDD Private Side: Partitions and Lineage.
3. Spark Application: Jobs, Stages and Tasks.

Spark Application: Jobs, Stages and Tasks

- A Spark user program must begin by declaring a **SparkContext** variable **sc**.

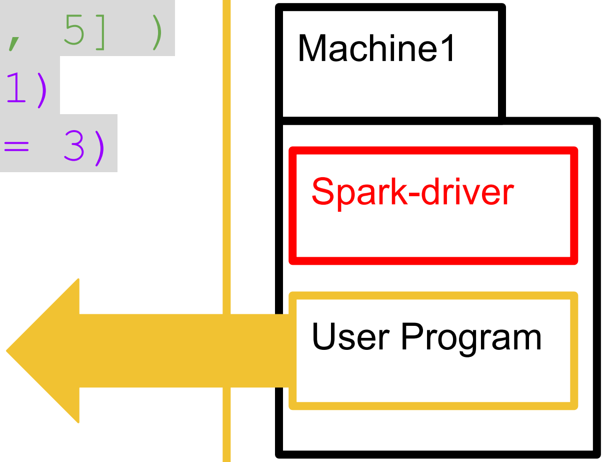
```
inputRDD = sc.parallelize( [ 1, 2, 3, 4, 5] )
mappedRDD = inputRDD.map(lambda x: x + 1)
solRDD = mappedRDD.filter(lambda x: x >= 3)
solRDD.persist( )
resVAL = filterRDD.count( )
solRDD.saveAsTextFile()
print(resVAL)
```



Spark Application: Jobs, Stages and Tasks

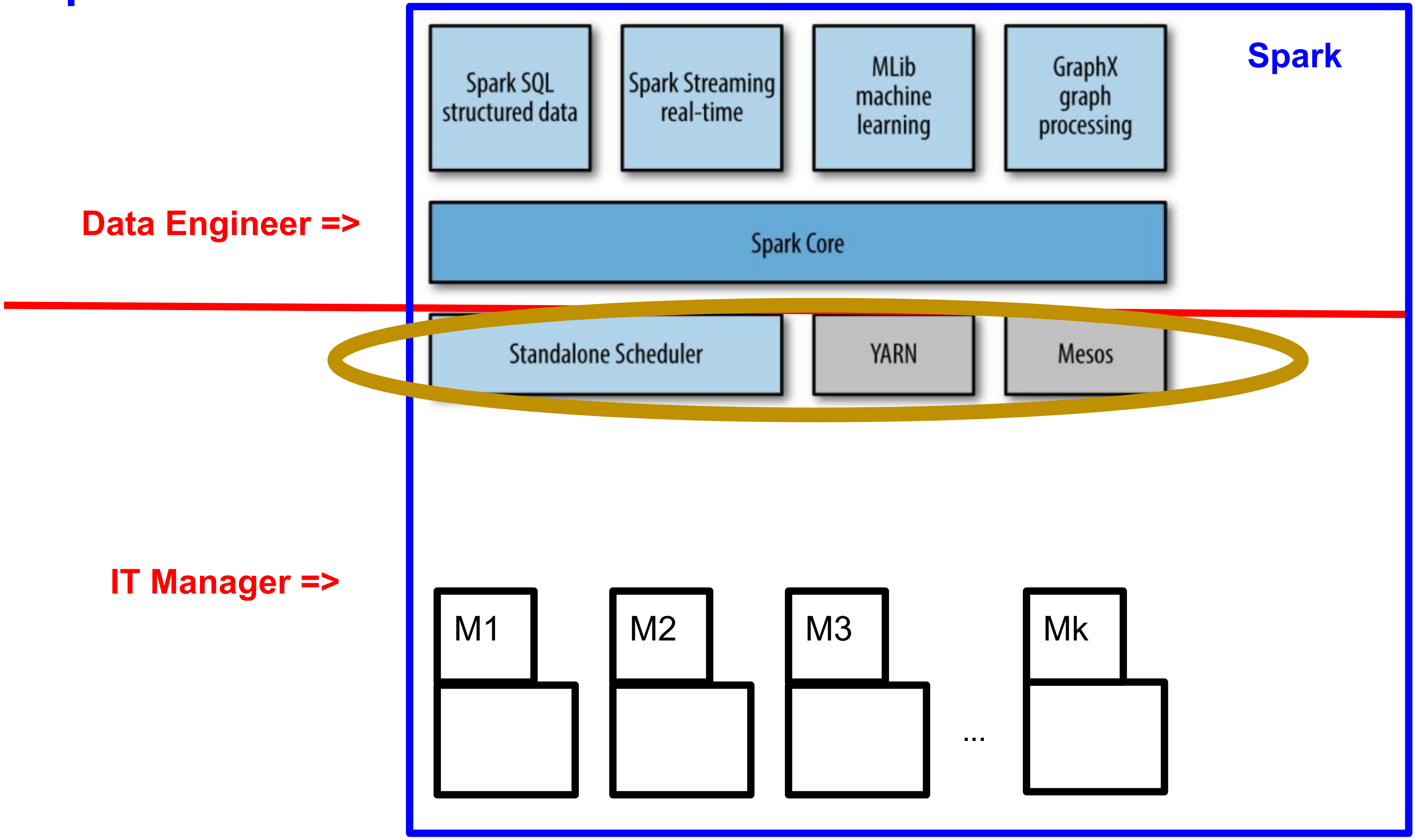
- A Spark user program must begin by declaring a **SparkContext** variable **sc**.

```
sc = pyspark.SparkContext.getOrCreate()
inputRDD = sc.parallelize( [ 1, 2, 3, 4, 5] )
mappedRDD = inputRDD.map(lambda x: x + 1)
solRDD = mappedRDD.filter(lambda x: x >= 3)
solRDD.persist( )
resVAL = filterRDD.count( )
solRDD.saveAsTextFile()
print(resVAL)
```



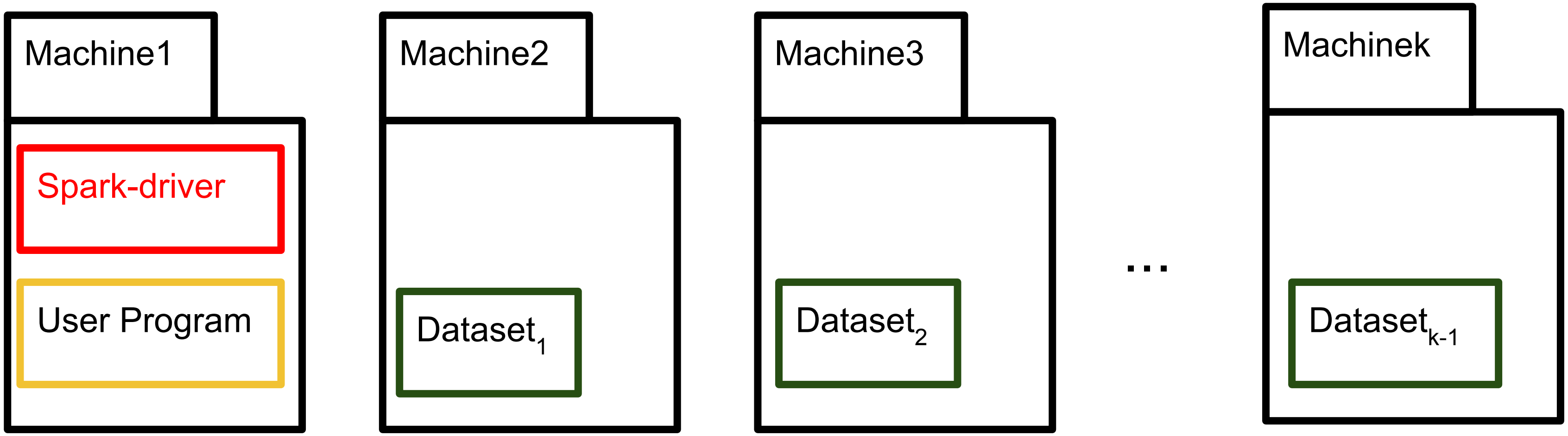
Spark Application: Jobs, Stages and Tasks

- This makes the **Spark driver** to ping the cluster manager for launching the **Spark executors** across the different machines.



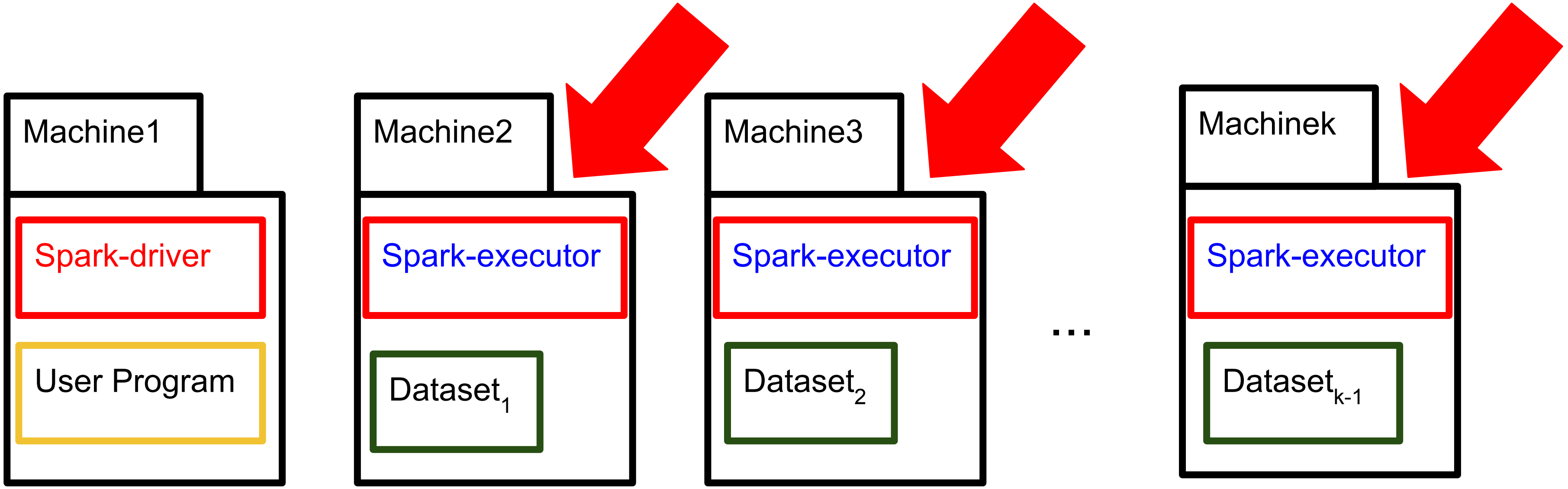
Spark Application: Jobs, Stages and Tasks

- This makes the **Spark driver** to ping the cluster manager for launching the **Spark executors** across the different machines.



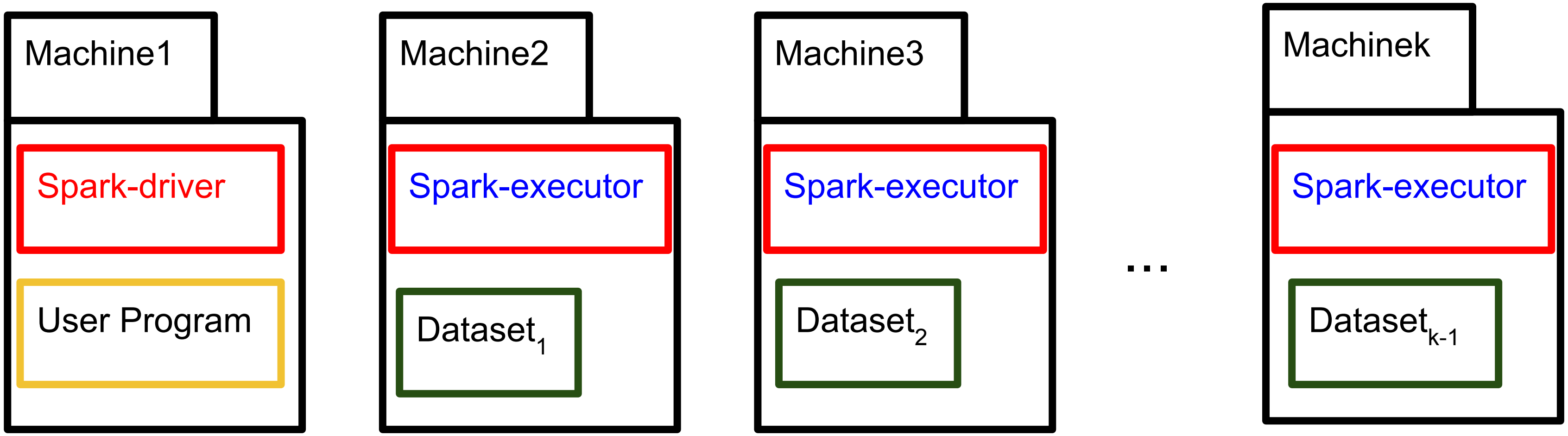
Spark Application: Jobs, Stages and Tasks

- This makes the **Spark driver** to ping the cluster manager for launching the **Spark executors** across the different machines.



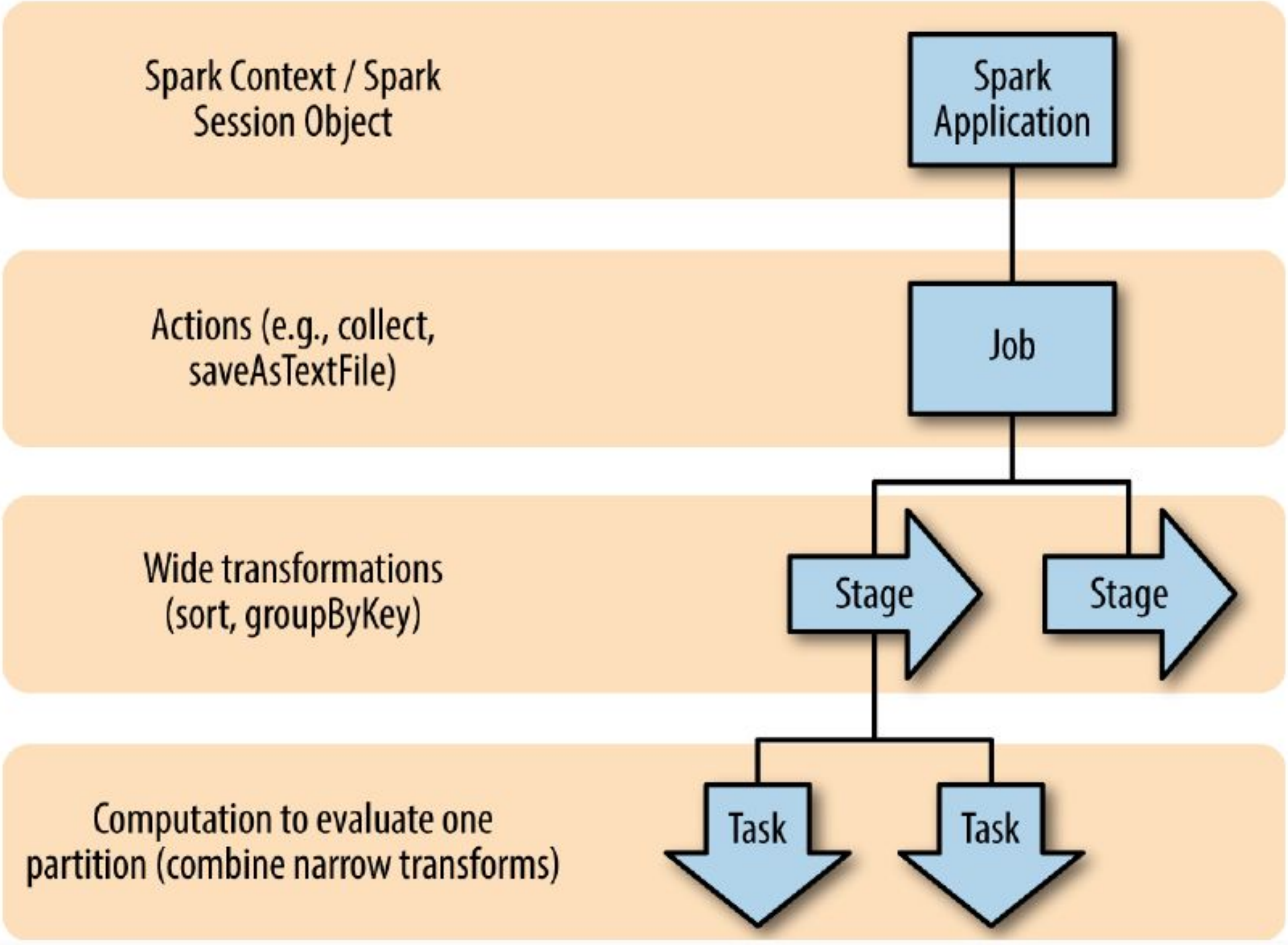
Spark Application: Jobs, Stages and Tasks

- This makes the **Spark driver** to ping the cluster manager for launching the **Spark executors** across the different machines.
- Each machine can host multiple Spark executors, but an executor cannot span multiple nodes.
- Likewise, as we have seen, each executor can host multiple partitions of an RDD, but a partition cannot be spread across multiple executors.



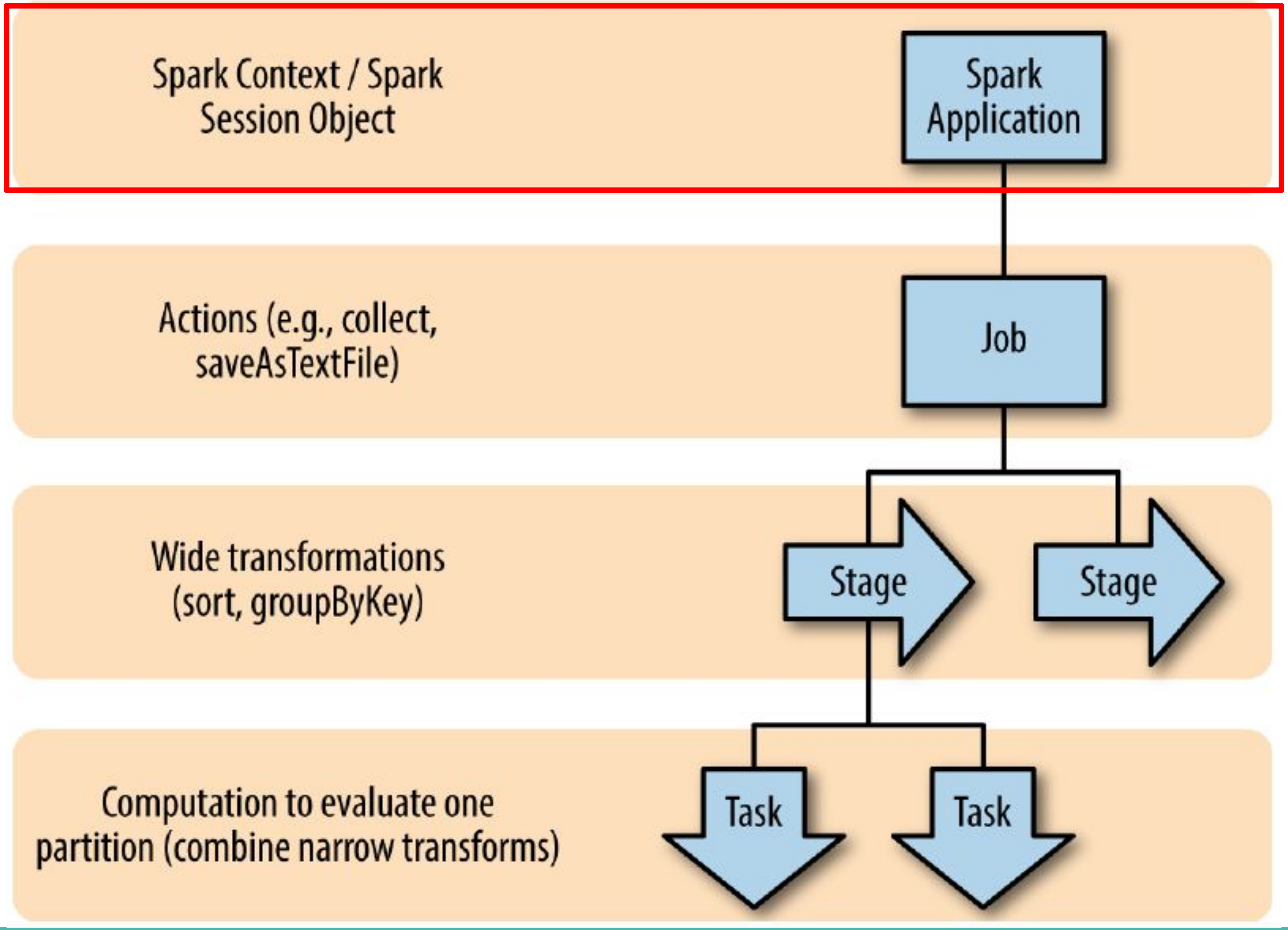
Spark Application: Jobs, Stages and Tasks

- There are 4 concepts we must be familiar with in order to understand the execution of a Spark user program: Application, Job, Stage and Task.


Spark Context / Spark Session Object
Spark Application
Actions (e.g., collect, saveAsTextFile)
Job
Wide transformations (sort, groupByKey)
Stage
Stage
Computation to evaluate one partition (combine narrow transforms)
Task
Task

Spark Application: Jobs, Stages and Tasks

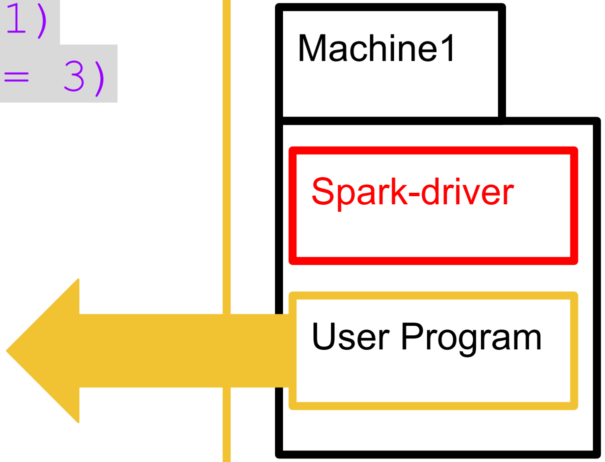
- A Spark Application corresponds to a Spark User program.


Spark Context / Spark Session Object
Spark Application
Actions (e.g., collect, saveAsTextFile)
Job
Wide transformations (sort, groupByKey)
Stage
Stage
Computation to evaluate one partition (combine narrow transforms)
Task
Task

Spark Application: Jobs, Stages and Tasks

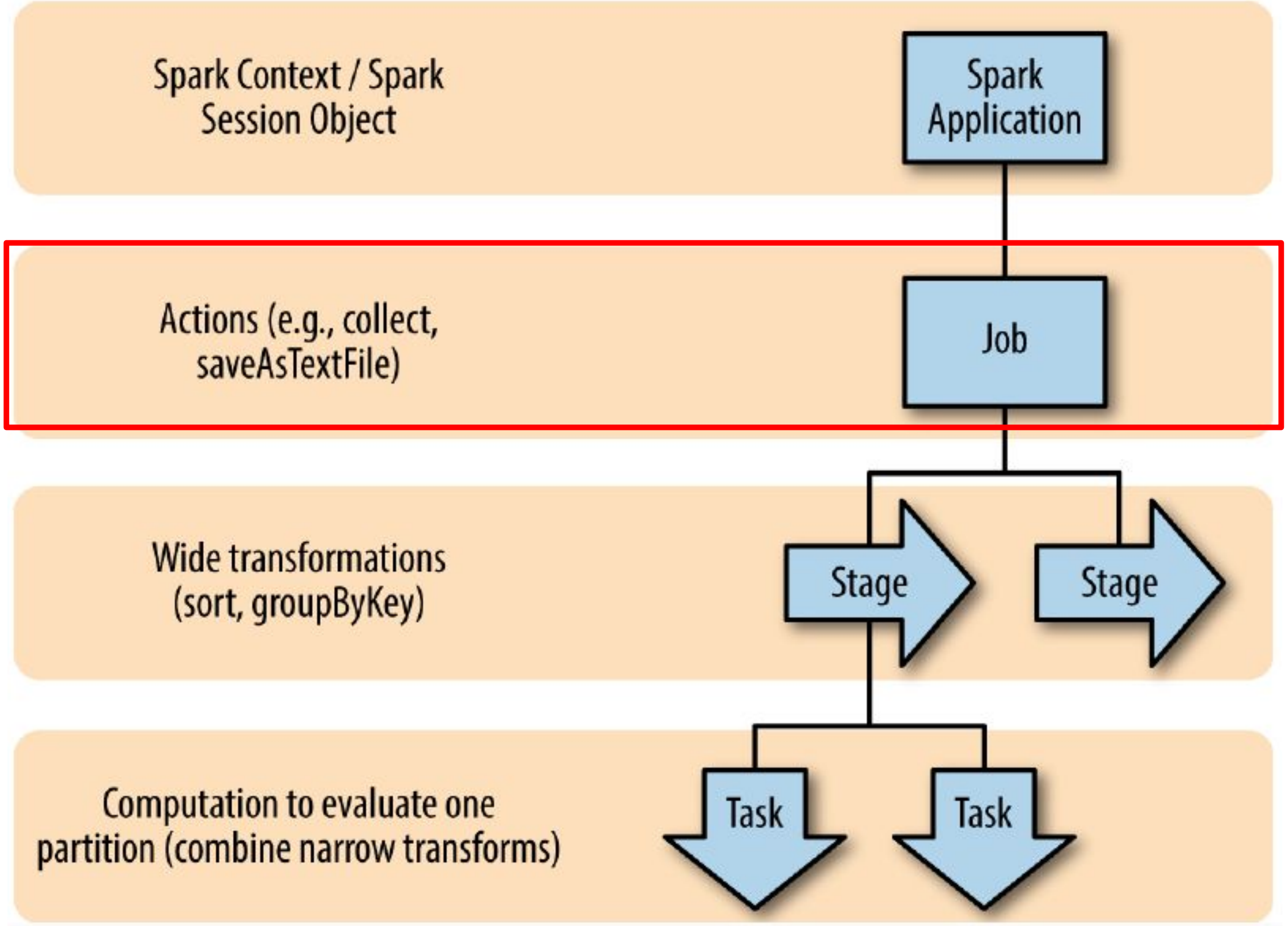
- A Spark Application corresponds to a Spark User program.

```
sc = pyspark.SparkContext.getOrCreate()
inputRDD = sc.parallelize( [ 1, 2, 3, 4, 5] )
mappedRDD = inputRDD.map(lambda x: x + 1)
solRDD = mappedRDD.filter(lambda x: x >= 3)
solRDD.persist( )
resVAL = filterRDD.count( )
solRDD.saveAsTextFile()
print(resVAL)
```



Spark Application: Jobs, Stages and Tasks

- A Spark Job corresponds to one **action** operation in the user program. Thus, given a user program, it leads to as many jobs as actions it contains.


Spark Context / Spark Session Object
Spark Application
Actions (e.g., collect, saveAsTextFile)
Job
Wide transformations (sort, groupByKey)
Stage
Stage
Computation to evaluate one partition (combine narrow transforms)
Task
Task

Spark Application: Jobs, Stages and Tasks

- A Spark Job corresponds to one **action** operation in the user program. Thus, given a user program, it leads to as many jobs as actions it contains.

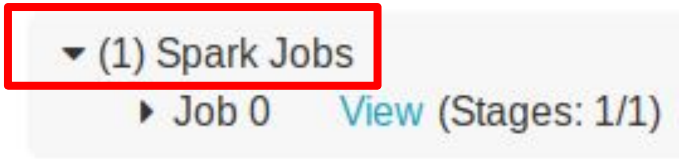
```
inputRDD   = sc.parallelize( [ 1, 2, 3, 4, 5 ] )
mappedRDD  = inputRDD.map( lambda elem : elem + 1 )
filteredRDD  = mappedRDD.filter( lambda elem : elem > 3 )
resVAL = filteredRDD.count()
```

Spark Application: Jobs, Stages and Tasks

- A Spark Job corresponds to one **action** operation in the user program. Thus, given a user program, it leads to as many jobs as actions it contains.

```
inputRDD  = sc.parallelize( [ 1, 2, 3, 4, 5 ] )
mappedRDD  = inputRDD.map( lambda elem : elem + 1 )
filteredRDD  = mappedRDD.filter( lambda elem : elem > 3 )
resVAL = filteredRDD.count()
```

- This program leads to 1 job.



▾ (1) Spark Jobs
▸ Job 0 View (Stages: 1/1)

- The result is 3.

3

Spark Application: Jobs, Stages and Tasks

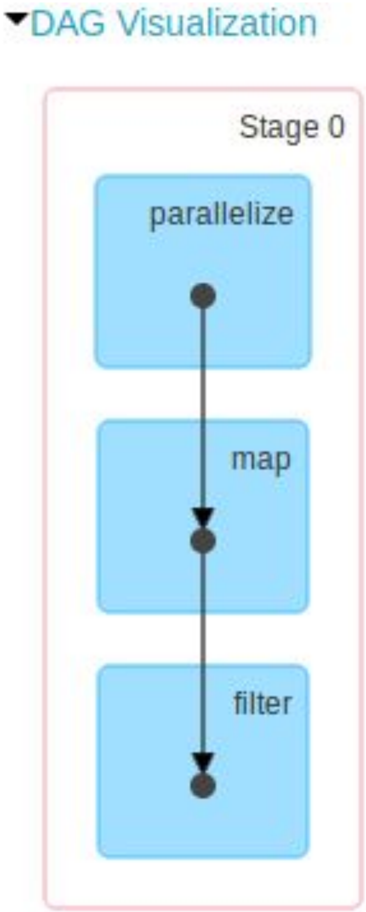
- Formally, the lineage definition is called the Direct Acyclic Graph (DAG). On it, the **action** operation is the leaf of the graph, and the **creation**/**transformation** operations are the intermediate nodes to get to it.

```
inputRDD  = sc.parallelize( [ 1, 2, 3, 4, 5 ] )
mappedRDD  = inputRDD.map( lambda elem : elem + 1 )
filteredRDD  = mappedRDD.filter( lambda elem :elem > 3 )
resVAL = filteredRDD.count()
```

Spark Application: Jobs, Stages and Tasks

- Formally, the lineage definition is called the Direct Acyclic Graph (DAG). On it, the **action** operation is the leaf of the graph, and the **creation**/**transformation** operations are the intermediate nodes to get to it.

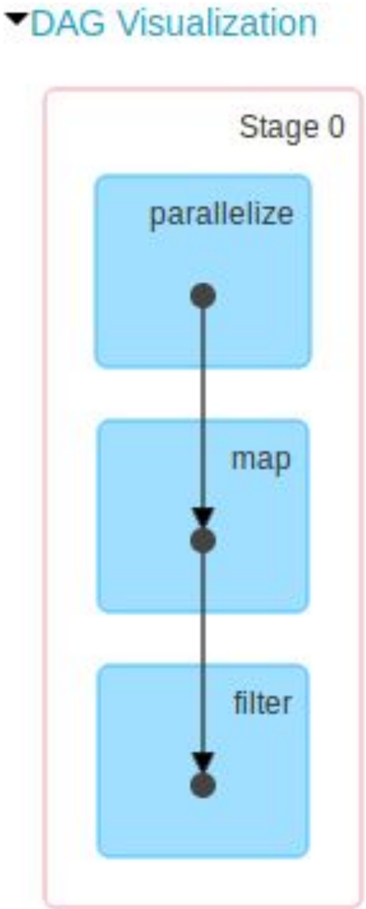
```
inputRDD    = sc.parallelize( [ 1, 2, 3, 4, 5 ] )
mappedRDD   = inputRDD.map( lambda elem : elem + 1 )
filteredRDD  = mappedRDD.filter( lambda elem :
                                              elem>3 )

resVAL = filteredRDD.count()
```



Spark Application: Jobs, Stages and Tasks

- Formally, the lineage definition is called the Direct Acyclic Graph (DAG). On it, the **action** operation is the leaf of the graph, and the **creation**/**transformation** operations are the intermediate nodes to get to it.

```
inputRDD   = sc.parallelize( [ 1, 2, 3, 4, 5 ] )
mappedRDD   = inputRDD.map( lambda elem : elem + 1 )
filteredRDD   = mappedRDD.filter( lambda elem :
                                           elem>3 )
resVAL = filteredRDD.count()
```

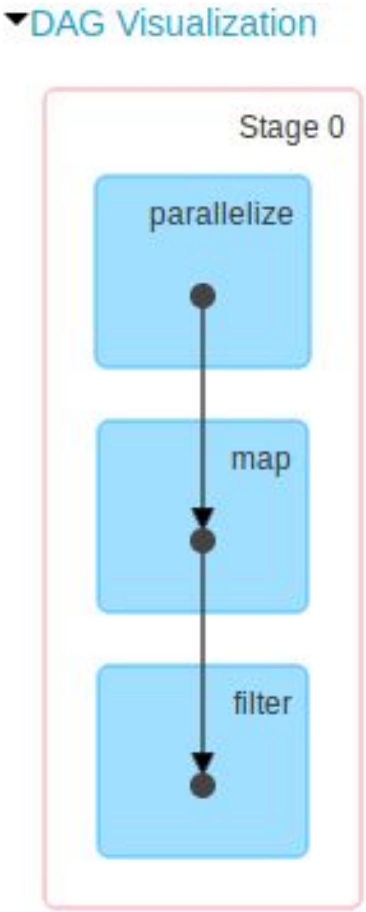


- As we can see, the **action** operation is not even represented in the DAG.

Spark Application: Jobs, Stages and Tasks

- Formally, the lineage definition is called the Direct Acyclic Graph (DAG). On it, the **action** operation is the leaf of the graph, and the **creation**/**transformation** operations are the intermediate nodes to get to it.

```
inputRDD   = sc.parallelize( [ 1, 2, 3, 4, 5 ] )
mappedRDD   = inputRDD.map( lambda elem : elem + 1 )
filteredRDD  = mappedRDD.filter( lambda elem :
                                              elem>3 )
resVAL = filteredRDD.count()
```



- As we can see, the **action** operation is not even represented in the DAG.
- Also, the DAG focuses in the RDD public side, representing RDDs as atomic variables (rather than as it internal partitions).

Spark Application: Jobs, Stages and Tasks

- Sometimes the 1 to 1 equivalence between **action** operations and jobs is not fully accurate.

Let’s use this program as an example:

```
inputRDD  = sc.parallelize( [ "Hello", "Hola", "Bonjour","Hello",
                                  "Bonjour", "Ciao", "Hello" ] )
pairWordsRDD = inputRDD.map( lambda x : (x, 1) )
countRDD = pairWordsRDD.reduceByKey( lambda x, y : x + y )
swapTupleRDD = countRDD.map( lambda x : (x[1], x[0]) )
filteredRDD = swapTupleRDD.filter( lambda x : x[0] > 1 )
solutionRDD = filteredRDD.sortByKey()
resVAL = solutionRDD.collect()
```

Spark Application: Jobs, Stages and Tasks

- Sometimes the 1 to 1 equivalence between **action** operations and jobs is not fully accurate.

Let’s use this program as an example:

- It first computes the word count from the list of words.
- Then, it filters out the ones appearing just once.
- Finally, it sorts them in increasing order by the amount of appearances.

```
inputRDD  = sc.parallelize( [ "Hello", "Hola", "Bonjour","Hello",
                                  "Bonjour", "Ciao", "Hello" ] )
pairWordsRDD = inputRDD.map( lambda x : (x, 1) )
countRDD = pairWordsRDD.reduceByKey( lambda x, y : x + y )
swapTupleRDD = countRDD.map( lambda x : (x[1], x[0]) )
filteredRDD = swapTupleRDD.filter( lambda x : x[0] > 1 )
solutionRDD = filteredRDD.sortByKey()
resVAL = solutionRDD.collect()
```

Spark Application: Jobs, Stages and Tasks

- Sometimes the 1 to 1 equivalence between **action** operations and jobs is not fully accurate.

Let's use this program as an example:

- It first computes the word count from the list of words.
- Then, it filters out the ones appearing just once.
- Finally, it sorts them in increasing order by the amount of appearances.

```
inputRDD  = sc.parallelize( [ "Hello", "Hola", "Bonjour","Hello",
                                   "Bonjour", "Ciao", "Hello" ] )
pairWordsRDD = inputRDD.map( lambda x : (x, 1) )
countRDD = pairWordsRDD.reduceByKey( lambda x, y : x + y )
swapTupleRDD = countRDD.map( lambda x : (x[1], x[0]) )
filteredRDD = swapTupleRDD.filter( lambda x : x[0] > 1 )
solutionRDD = filteredRDD.sortByKey()
resVAL = solutionRDD.collect()
```

Spark Application: Jobs, Stages and Tasks

- Sometimes the 1 to 1 equivalence between **action** operations and jobs is not fully accurate.

Let's use this program as an example:

- It first computes the word count from the list of words.
- Then, it filters out the ones appearing just once.
- Finally, it sorts them in increasing order by the amount of appearances.

```
inputRDD  = sc.parallelize( [ "Hello", "Hola", "Bonjour","Hello",
                                  "Bonjour", "Ciao", "Hello" ] )
pairWordsRDD = inputRDD.map( lambda x : (x, 1) )
countRDD = pairWordsRDD.reduceByKey( lambda x, y : x + y )
swapTupleRDD = countRDD.map( lambda x : (x[1], x[0]) )
filteredRDD = swapTupleRDD.filter( lambda x : x[0] > 1 )
solutionRDD = filteredRDD.sortByKey()
resVAL = solutionRDD.collect()
```

Spark Application: Jobs, Stages and Tasks

- Sometimes the 1 to 1 equivalence between **action** operations and jobs is not fully accurate.

Let’s use this program as an example:

  - It first computes the word count from the list of words.
  - Then, it filters out the ones appearing just once.
  - Finally, it sorts them in increasing order by the amount of appearances.

```
inputRDD  = sc.parallelize( [ "Hello", "Hola", "Bonjour","Hello",
                                   "Bonjour", "Ciao", "Hello" ] )
pairWordsRDD = inputRDD.map( lambda x : (x, 1) )
countRDD = pairWordsRDD.reduceByKey( lambda x, y : x + y )
swapTupleRDD = countRDD.map( lambda x : (x[1], x[0]) )
filteredRDD = swapTupleRDD.filter( lambda x : x[0] > 1 )
solutionRDD = filteredRDD.sortByKey()
resVAL = solutionRDD.collect()
```

Spark Application: Jobs, Stages and Tasks

- Sometimes the 1 to 1 equivalence between **action** operations and jobs is not fully accurate.

Let’s use this program as an example:

  - The program has one single **action** operation, to collect the final list of words for displaying it by the screen.

```
inputRDD  = sc.parallelize( [ "Hello", "Hola", "Bonjour","Hello",
                                  "Bonjour", "Ciao", "Hello" ] )
pairWordsRDD = inputRDD.map( lambda x : (x, 1) )
countRDD = pairWordsRDD.reduceByKey( lambda x, y : x + y )
swapTupleRDD = countRDD.map( lambda x : (x[1], x[0]) )
filteredRDD = swapTupleRDD.filter( lambda x : x[0] > 1 )
solutionRDD = filteredRDD.sortByKey()
resVAL = solutionRDD.collect()
```

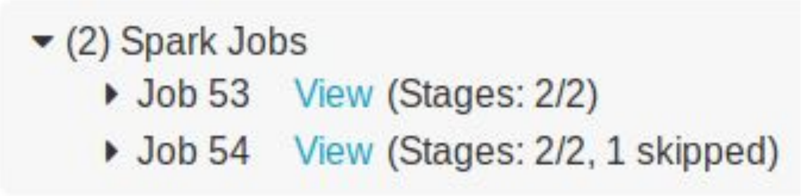
Spark Application: Jobs, Stages and Tasks

- Sometimes the 1 to 1 equivalence between **action** operations and jobs is not fully accurate.

Let's use this program as an example:

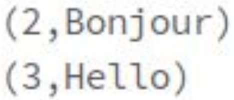
  - The program has one single **action** operation, to collect the final list of words for displaying it by the screen.

- As we can see, the program leads to 2 jobs, even if it only has 1 **action**.



```
▾ (2) Spark Jobs
    ▸ Job 53   View (Stages: 2/2)
    ▸ Job 54   View (Stages: 2/2, 1 skipped)
```

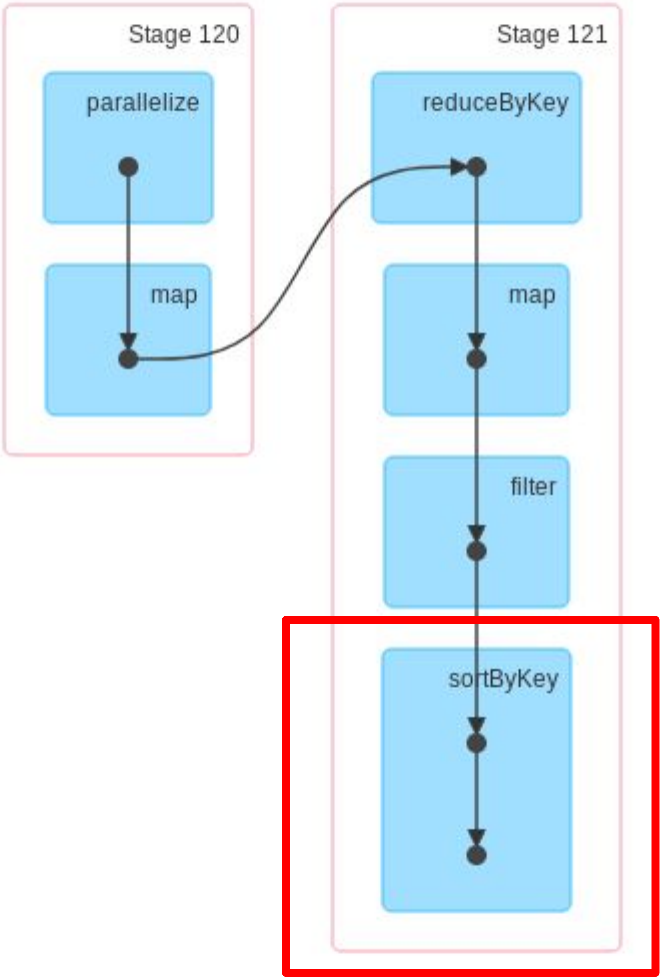
- The result is the 2 elements.



```
(2,Bonjour)
(3,Hello)
```

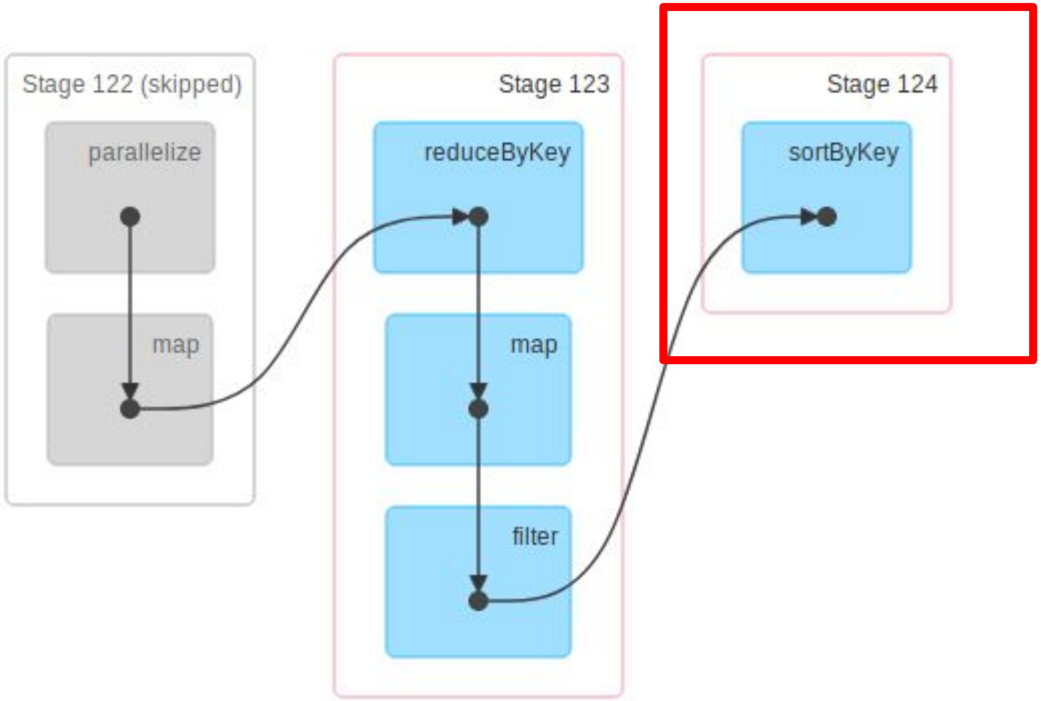
Spark Application: Jobs, Stages and Tasks

- However, the 2 jobs are indeed related:
  - Job 53 seems to fail to break sortByKey( ) into a new stage, as it should.



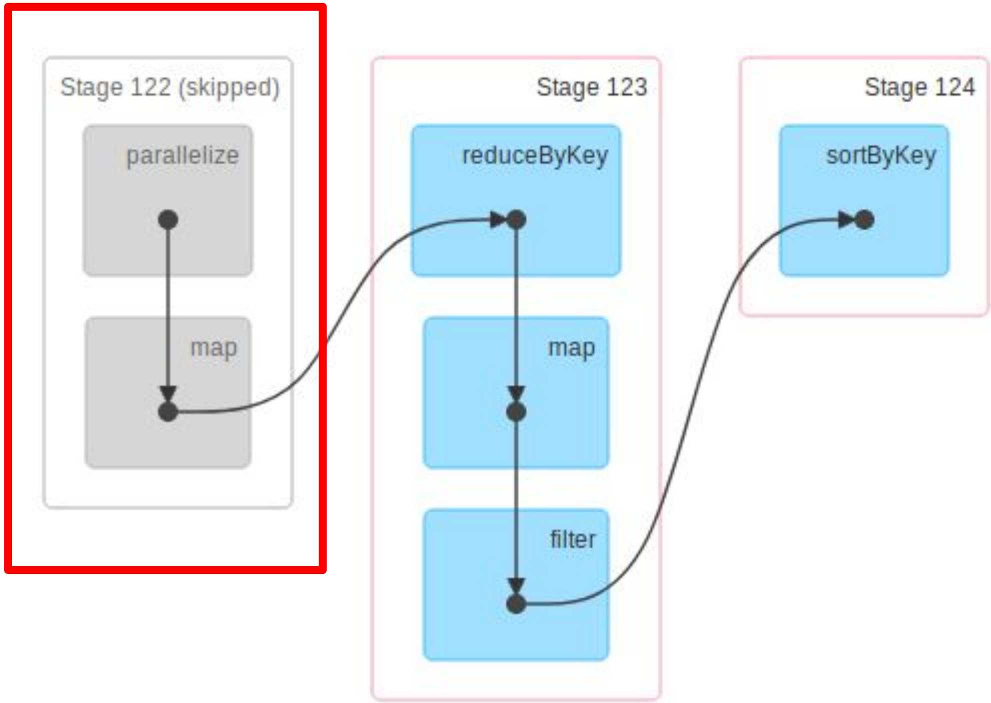
Spark Application: Jobs, Stages and Tasks

- However, the 2 jobs are indeed related:
  - Job 54 takes over from Job 53 to finally break the operation into such desired new stage.



Spark Application: Jobs, Stages and Tasks

- However, the 2 jobs are indeed related:
  - As Job 54 is taking over from Job 53 it can indeed skip previous stages computed by it.



Spark Application: Jobs, Stages and Tasks

- Sometimes the 1 to 1 equivalence between **action** operations and jobs is not fully accurate.

Let's modify a bit the program used before:

```
inputRDD  = sc.parallelize( [ "Hello", "Hola", "Bonjour","Hello",
                                   "Bonjour", "Ciao", "Hello" ] )
pairWordsRDD = inputRDD.map( lambda x : (x, 1) )
countRDD = pairWordsRDD.reduceByKey( lambda x, y : x + y )
swapTupleRDD = countRDD.map( lambda x : (x[1], x[0]) )
filteredRDD = swapTupleRDD.filter( lambda x : x[0] > 1 )
solutionRDD = filteredRDD.sortByKey()
solutionRDD.persist()
resVAL1 = solutionRDD.collect()
resVAL2 = solutionRDD.count()
```

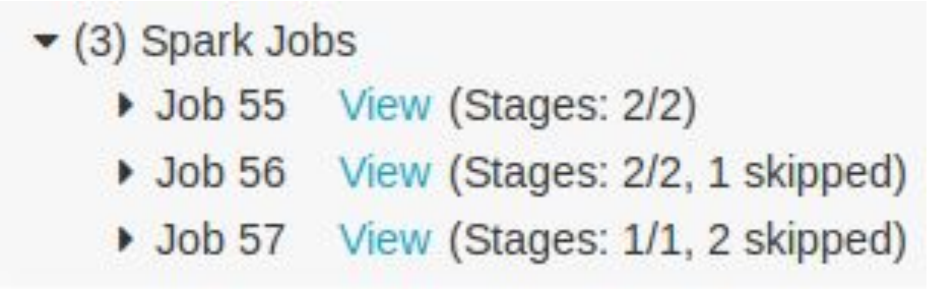
As we can see, the program now has 2 **action** operations, and has to **persist** solutionRDD as it be used both to **collect** and **count** its elements.

Spark Application: Jobs, Stages and Tasks

- Sometimes the 1 to 1 equivalence between **action** operations and jobs is not fully accurate.

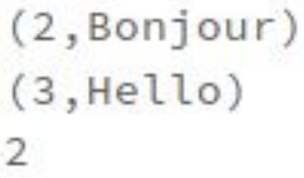
Let's modify a bit the program used before:

- As we can see, the program leads to 3 jobs, even if it only has 2 **actions**.



```
▾ (3) Spark Jobs
    ▸ Job 55   View (Stages: 2/2)
    ▸ Job 56   View (Stages: 2/2, 1 skipped)
    ▸ Job 57   View (Stages: 1/1, 2 skipped)
```

- The result is:



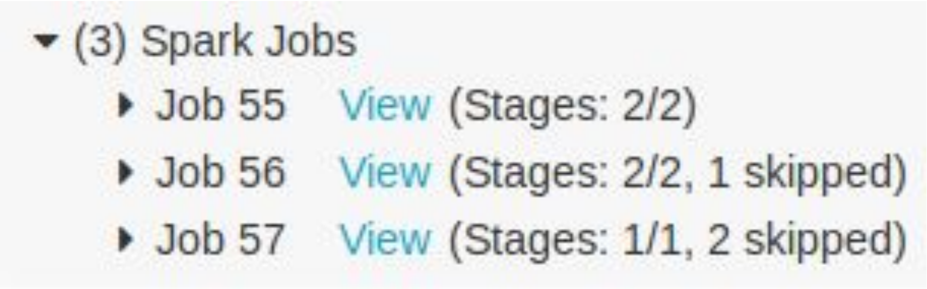
```
(2,Bonjour)
(3,Hello)
2
```

Spark Application: Jobs, Stages and Tasks

- Sometimes the 1 to 1 equivalence between **action** operations and jobs is not fully accurate.

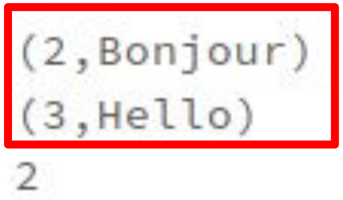
Let's modify a bit the program used before:

- As we can see, the program leads to 3 jobs, even if it only has 2 **actions**.



▾ (3) Spark Jobs
- ▸ Job 55 View (Stages: 2/2)
- ▸ Job 56 View (Stages: 2/2, 1 skipped)
- ▸ Job 57 View (Stages: 1/1, 2 skipped)

- The result is:
  - the 2 elements themselves
  - the count of elements



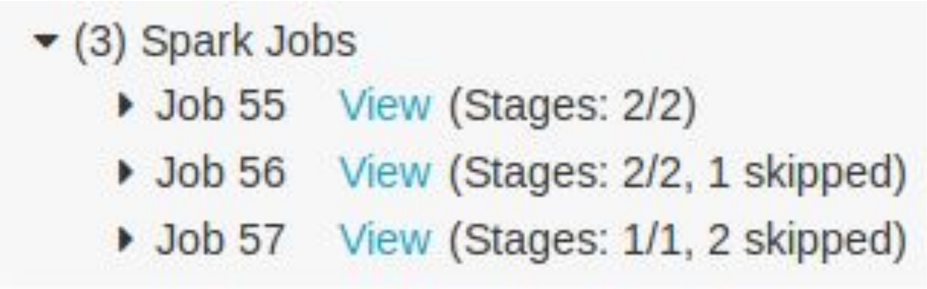
```
(2,Bonjour)
(3,Hello)
2
```

Spark Application: Jobs, Stages and Tasks

- Sometimes the 1 to 1 equivalence between **action** operations and jobs is not fully accurate.

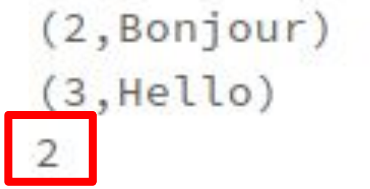
Let's modify a bit the program used before:

- As we can see, the program leads to 3 jobs, even if it only has 2 **actions**.



```
▾ (3) Spark Jobs
    ▸ Job 55   View (Stages: 2/2)
    ▸ Job 56   View (Stages: 2/2, 1 skipped)
    ▸ Job 57   View (Stages: 1/1, 2 skipped)
```

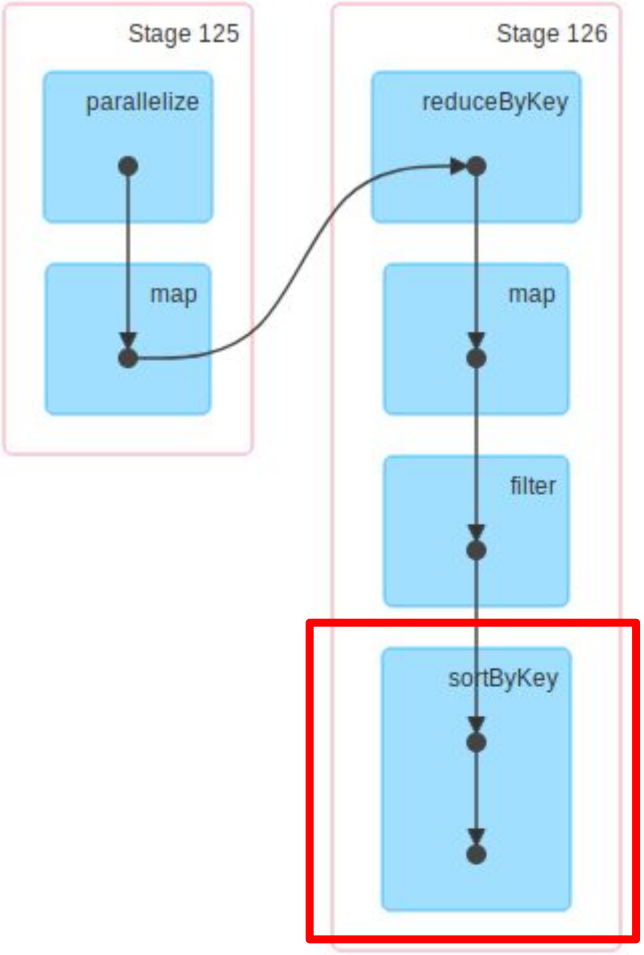
- The result is:
    - the 2 elements themselves
    - the count of elements



```
(2,Bonjour)
(3,Hello)
2
```

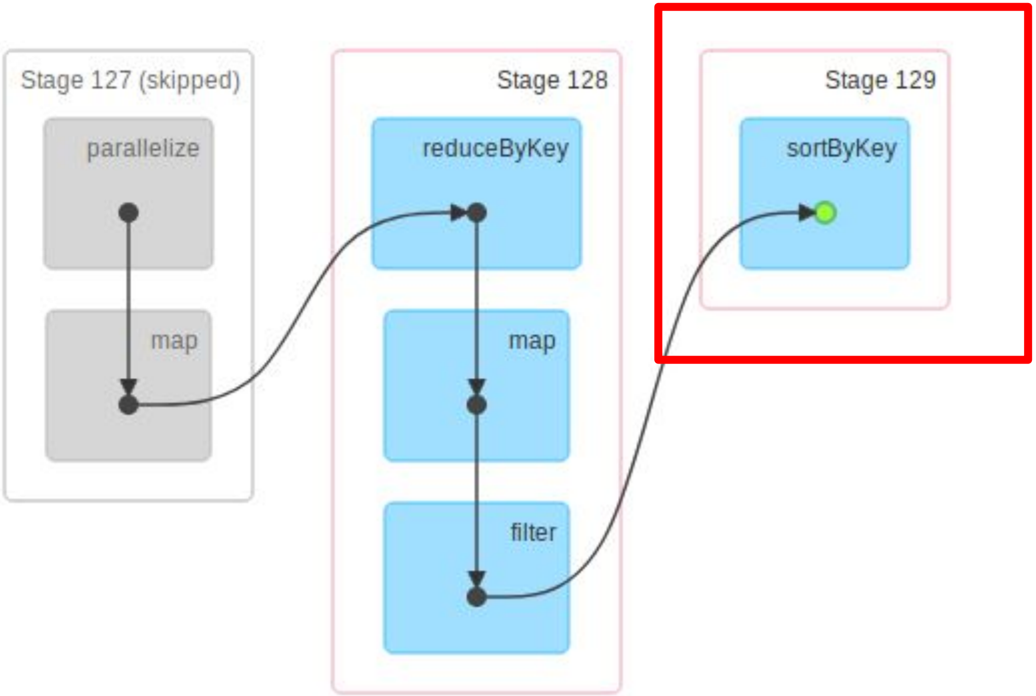
Spark Application: Jobs, Stages and Tasks

- However, again the first 2 jobs are indeed related and can be merged into one:
  - Job 55 seems to fail to break sortByKey( ) into a new stage, as it should.



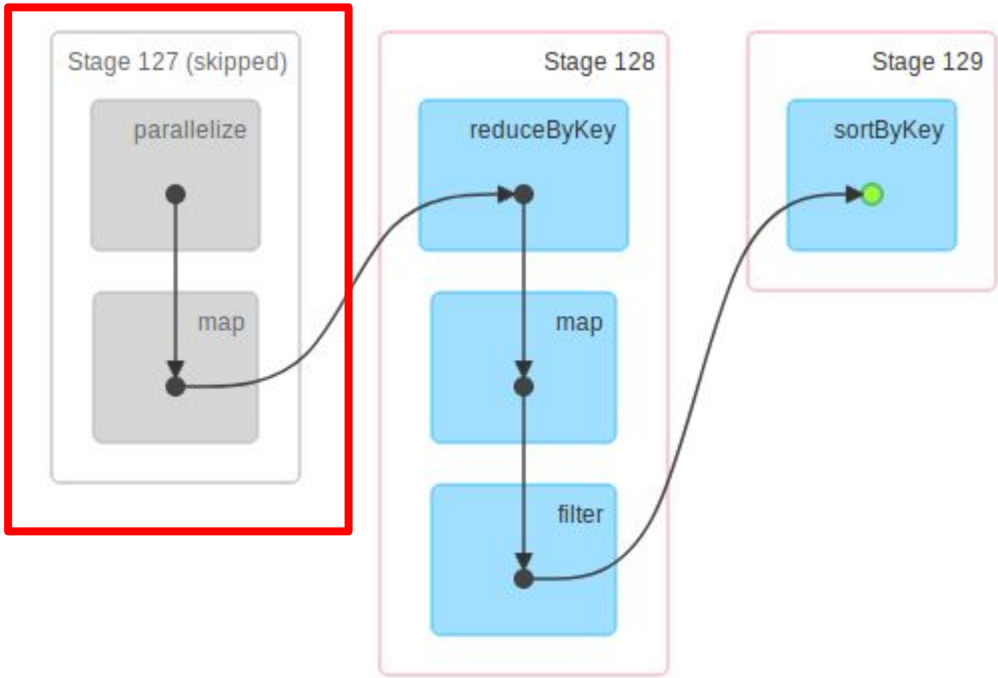
Spark Application: Jobs, Stages and Tasks

- However, again the first 2 jobs are indeed related and can be merged into one:
  - Job 56 takes over from Job 55 to finally break the operation into such desired new stage.



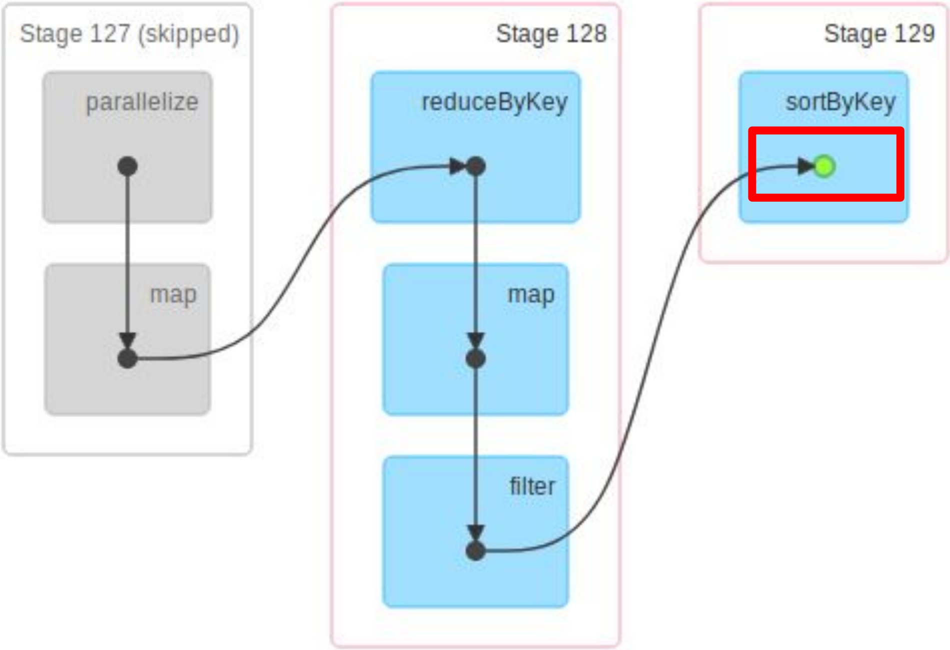
Spark Application: Jobs, Stages and Tasks

- However, again the first 2 jobs are indeed related and can be merged into one:
  - As Job 56 is taking over from Job 55 it can indeed skip previous stages computed by it.



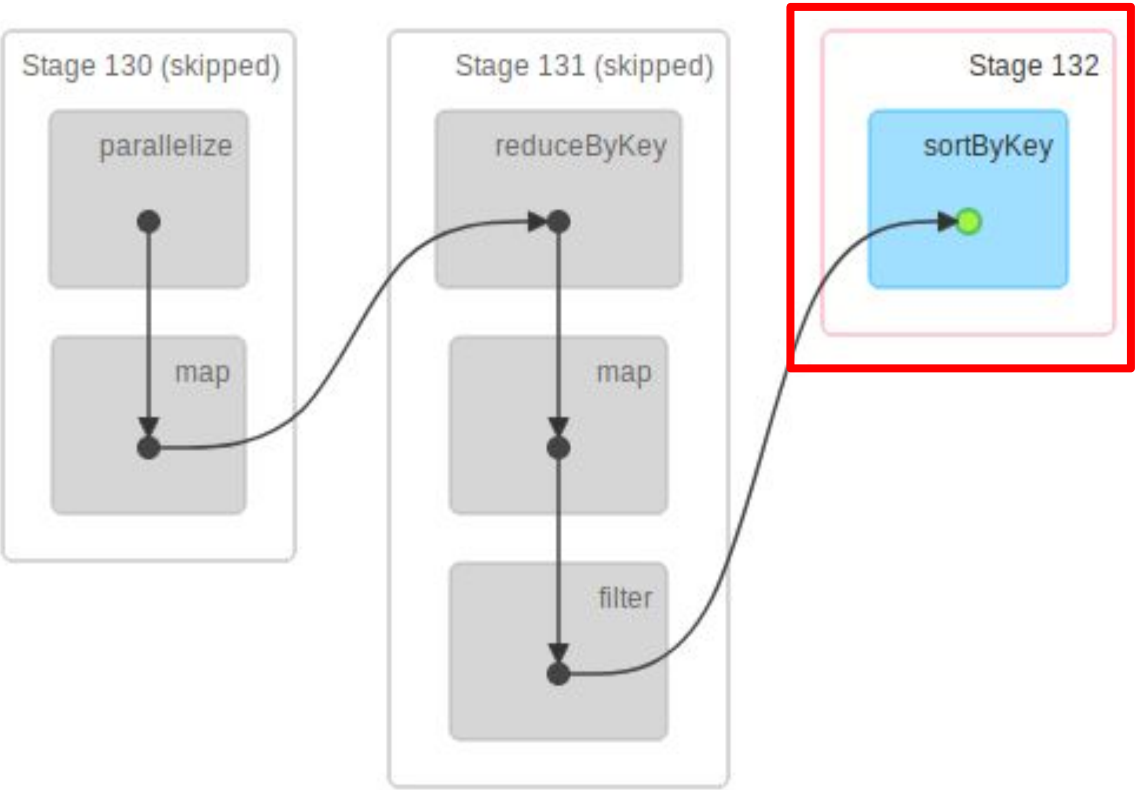
Spark Application: Jobs, Stages and Tasks

- However, again the first 2 jobs are indeed related and can be merged into one:
  - Finally, as we can see, the RDD solutionRDD created by **sortByKey** in Job 56 is **persisted**, which is indicated with a green circle.



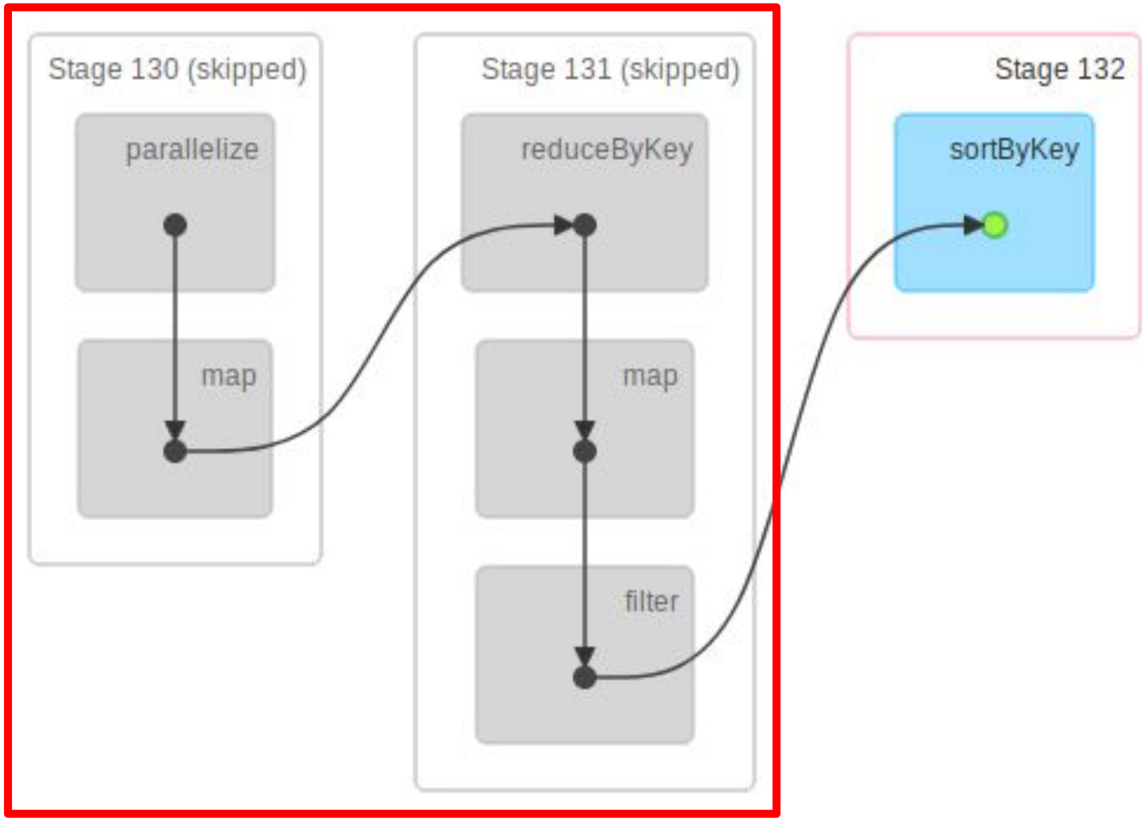
Spark Application: Jobs, Stages and Tasks

- Finally, Job 57 is in charge of the second **action** operation, **count**:
  - The Job takes over from the **persisted** solutionRDD to compute **count**.



Spark Application: Jobs, Stages and Tasks

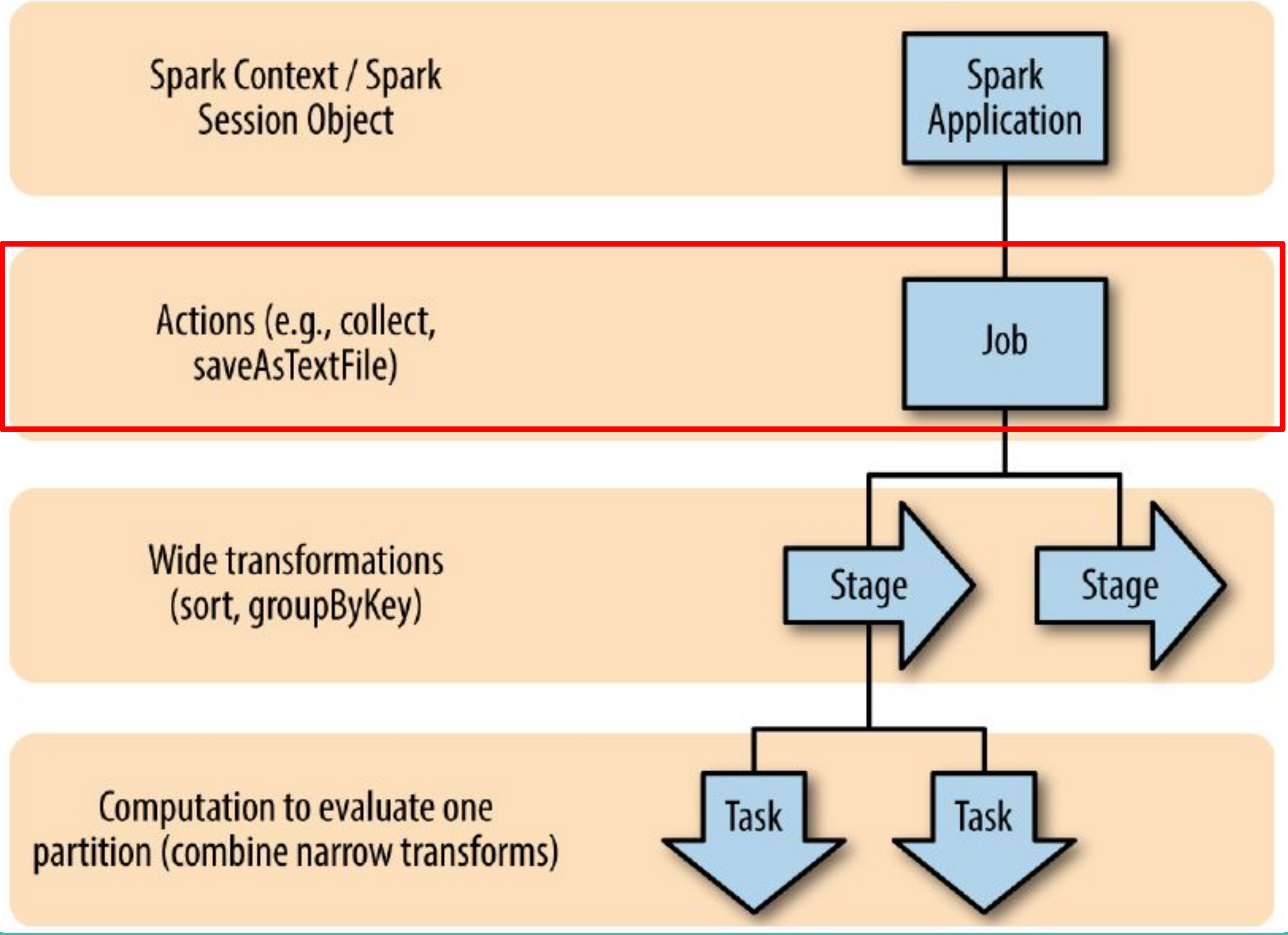
- Finally, Job 57 is in charge of the second **action** operation, **count**:
  - The Job takes over from the **persisted** solutionRDD to compute **count**. Thus, it can skip the previous stages accomplished by Jobs 55 and 56.



Spark Application: Jobs, Stages and Tasks

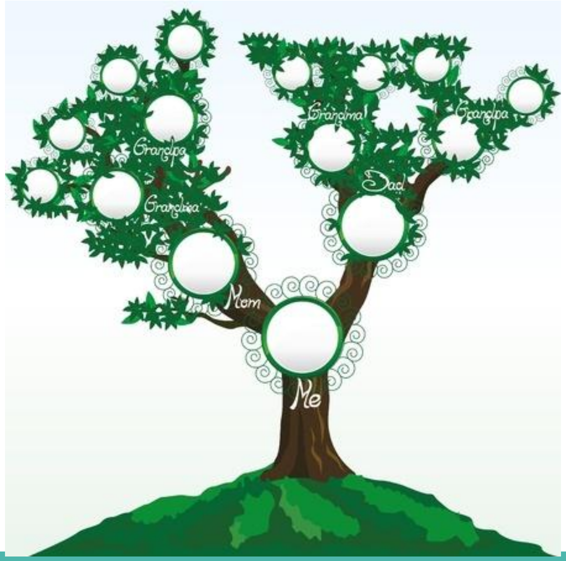
As previously mentioned, both the Jobs and the DAG generated for them seem to reason at the level of the RDD public side...

Spark Application: Jobs, Stages and Tasks


Spark Context / Spark Session Object
Spark Application
Actions (e.g., collect, saveAsTextFile)
Job
Wide transformations (sort, groupByKey)
Stage
Stage
Computation to evaluate one partition (combine narrow transforms)
Task
Task

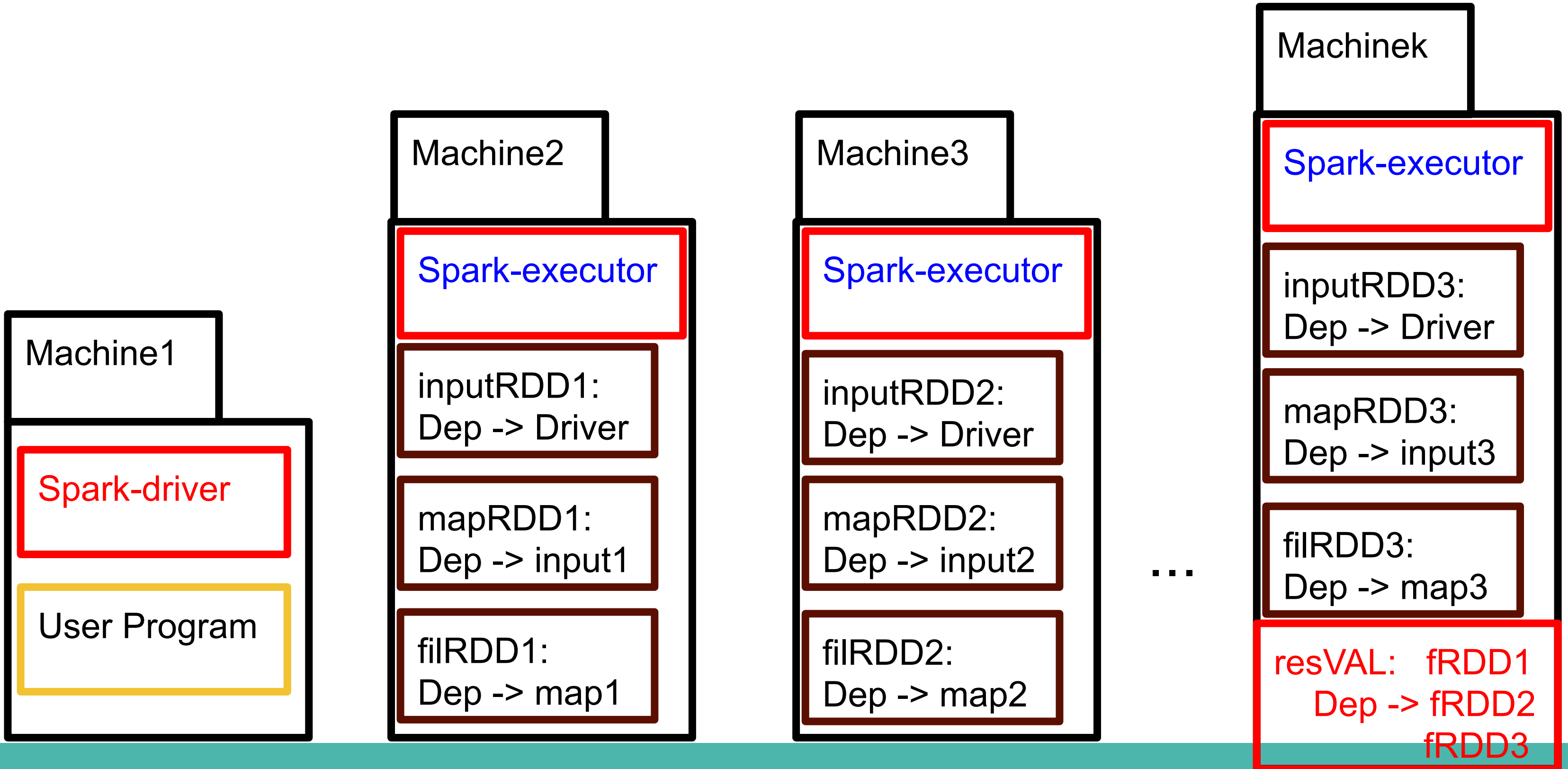
Spark Application: Jobs, Stages and Tasks

...however, as we know RDD are internally represented via partitions and lineage...



Spark Application: Jobs, Stages and Tasks

```
inputRDD  = sc.parallelize( [ 1, 2, 3, 4, 5 ] )
mappedRDD  = inputRDD.map( lambda elem : elem + 1 )
filteredRDD  = mappedRDD.filter( lambda elem : elem > 3 )
resVAL = filteredRDD.count()
```

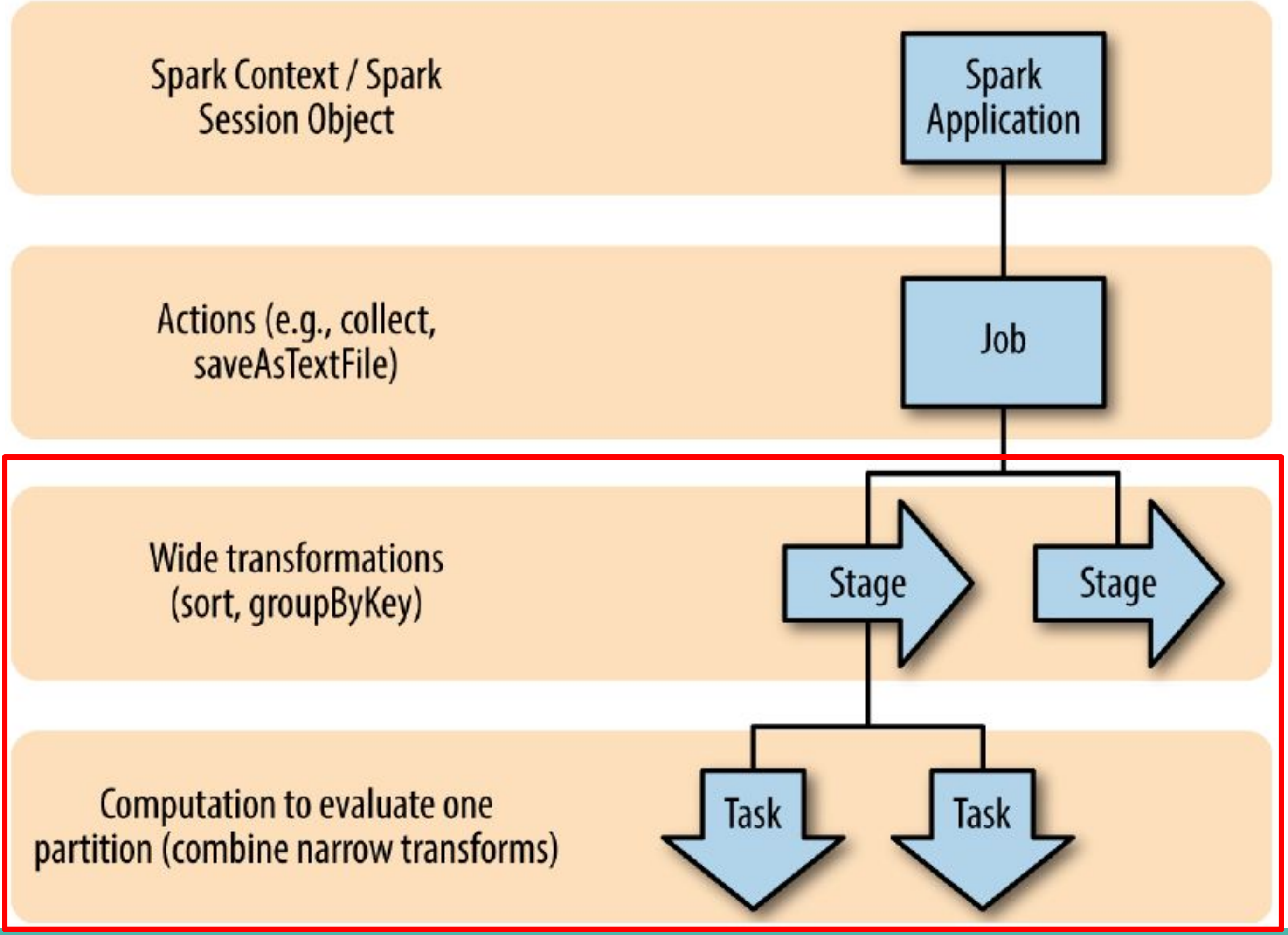


Spark Application: Jobs, Stages and Tasks

…thus, the DAG is passed to the TaskScheduler, who translate the rationale to the internal representation of RDDs via Stages and Tasks.

Spark Application: Jobs, Stages and Tasks

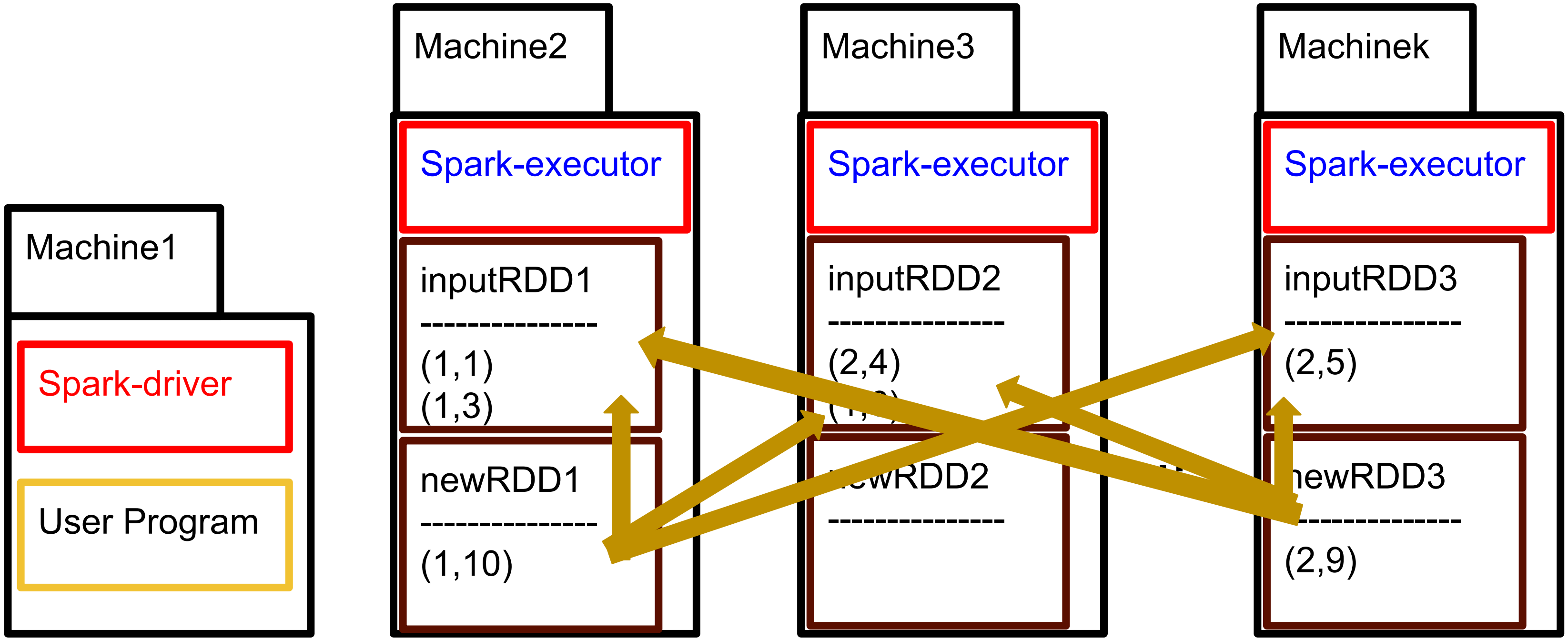
- The TaskScheduler must distinguish between the narrow and wide operations of the DAG.


Spark Context / Spark Session Object
Spark Application
Actions (e.g., collect, saveAsTextFile)
Job
Wide transformations (sort, groupByKey)
Stage
Stage
Computation to evaluate one partition (combine narrow transforms)
Task
Task

Spark Application: Jobs, Stages and Tasks

- As we have seen, wide operations require the shuffling of data, and thus network communication among the executor processes to perform the computation.

```
inputRDD  = sc.parallelize([(1,1), (2,4), (1,3), (2,5), (1,6)])
newRDD = inputRDD.reduceByKey( lambda x, y : x + y )
```

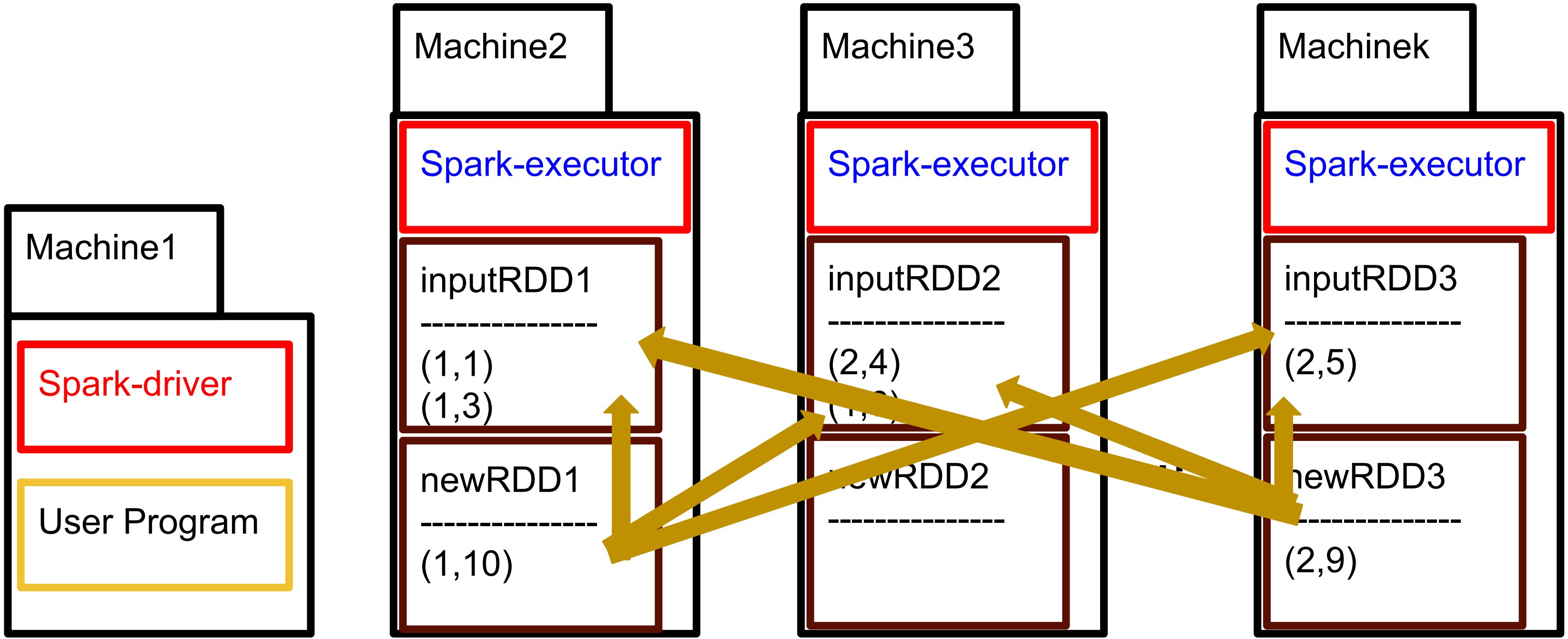


Spark Application: Jobs, Stages and Tasks

- As we have seen, wide operations require the shuffling of data, and thus network communication among the executor processes to perform the computation.

```
inputRDD  = sc.parallelize([(1,1), (2,4), (1,3), (2,5), (1,6)])
newRDD = inputRDD.reduceByKey( lambda x, y : x + y )
```

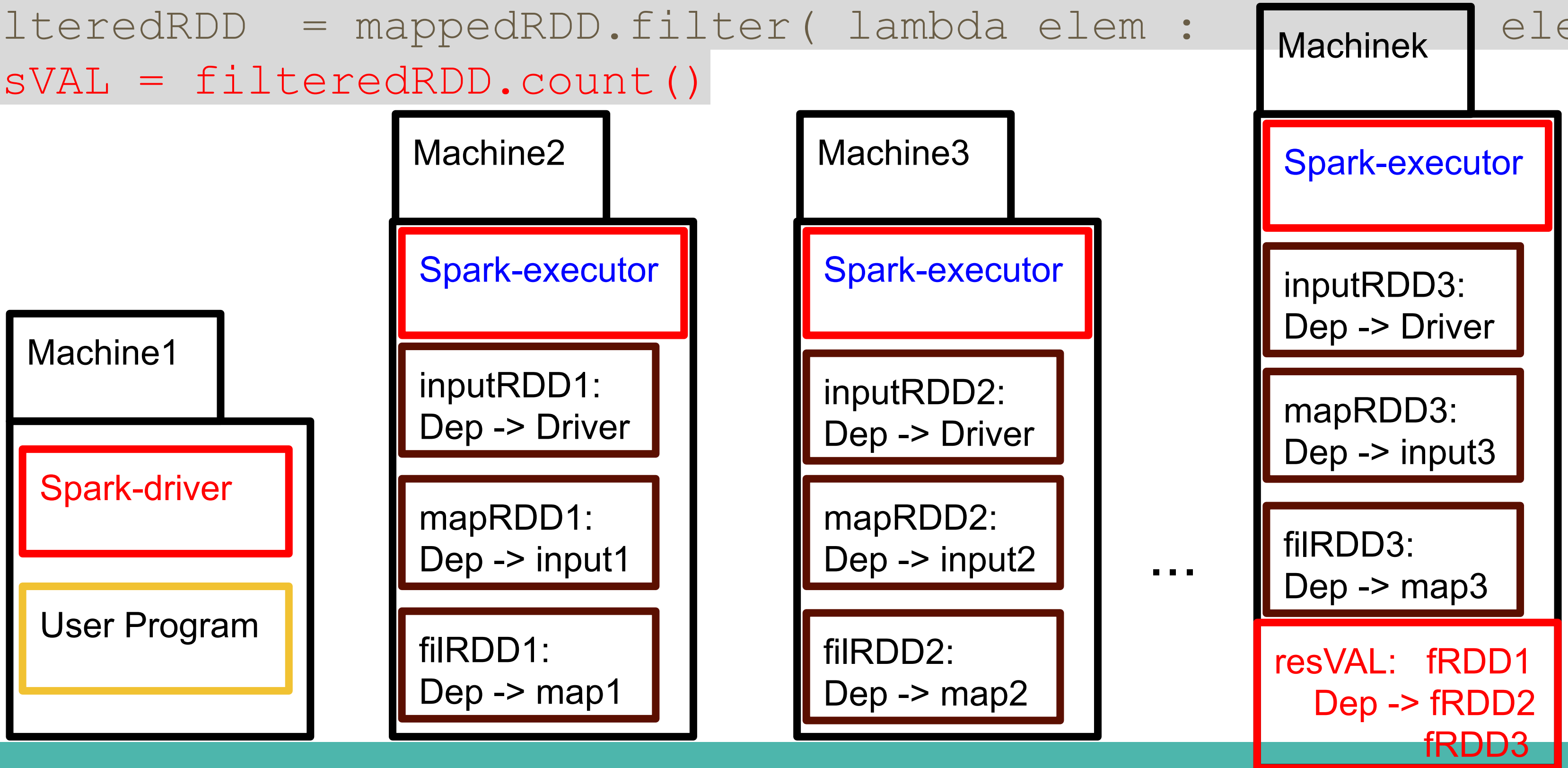
The DAG and the TaskScheduler refer to each of these wide operations as a stage.



Spark Application: Jobs, Stages and Tasks

- While this DAG is presented at a high level (treating RDDs as atomic variables), it has indeed all the lineage details we have seen in the previous section (so it can indeed reason at a partition level per RDD).

```
inputRDD  = sc.parallelize( [ 1, 2, 3, 4, 5 ] )
mappedRDD  = inputRDD.map( lambda elem : elem + 1 )
filteredRDD  = mappedRDD.filter( lambda elem :  elem>3 )
resVAL = filteredRDD.count()
```



Machine1

Spark-driver

User Program

Machine2

Spark-executor

inputRDD1: Dep -> Driver

mapRDD1: Dep -> input1

filRDD1: Dep -> map1

Machine3

Spark-executor

inputRDD2: Dep -> Driver

mapRDD2: Dep -> input2

filRDD2: Dep -> map2

...

Machinek

Spark-executor

inputRDD3: Dep -> Driver

mapRDD3: Dep -> input3

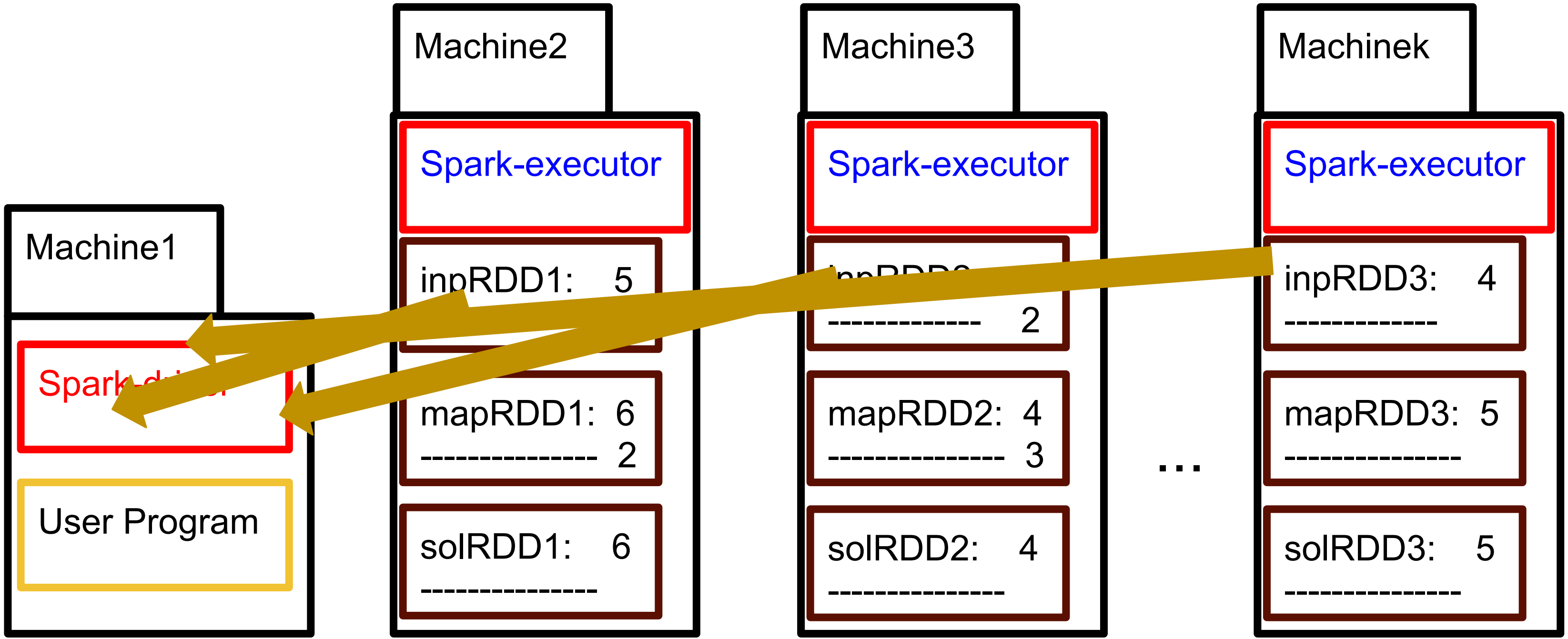
filRDD3: Dep -> map3

resVAL: fRDD1 Dep -> fRDD2 fRDD3

Spark Application: Jobs, Stages and Tasks

- On the contrary, narrow operations are performed locally, in a partition basis (with the executor process operating on it without any external communication).

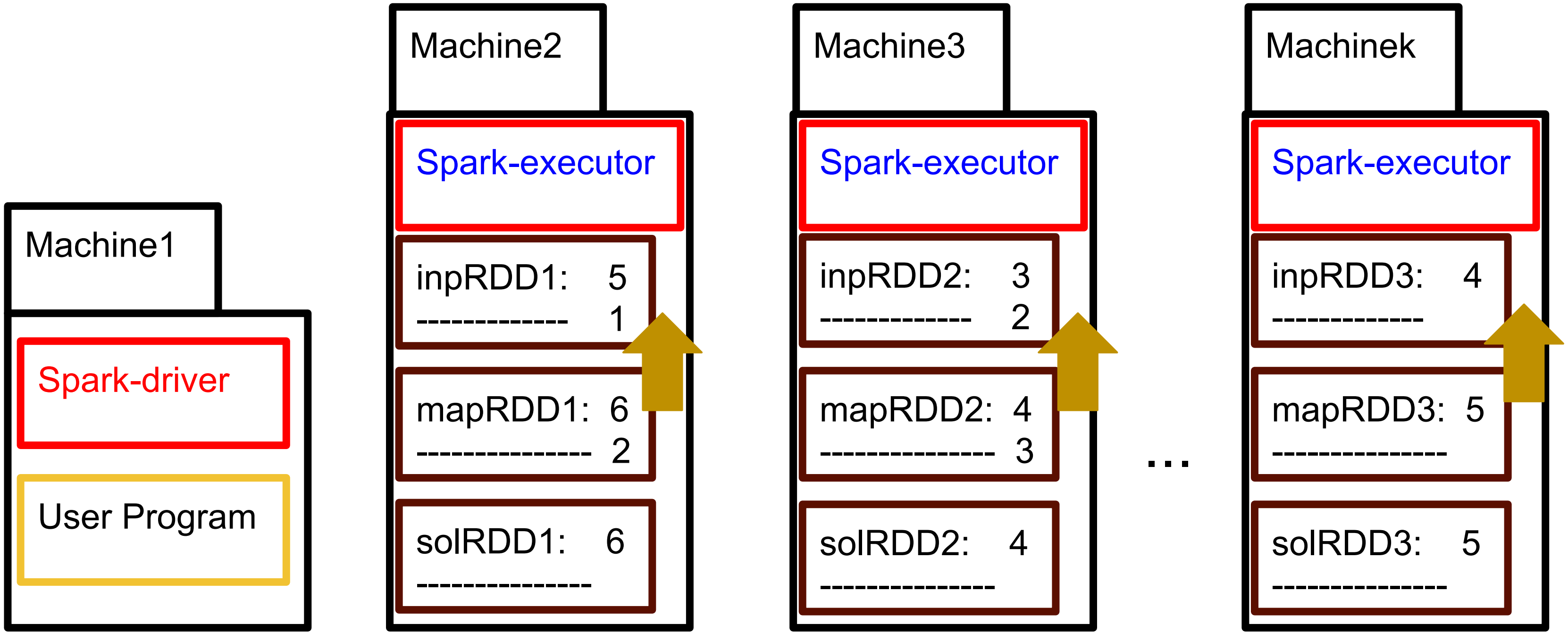
```
inputRDD  = sc.parallelize( [ 1, 2, 3, 4, 5 ] )
mapRDD = inputRDD.map( lambda elem : elem + 1 )
solRDD = mapRDD.filter( lambda elem : elem > 3 )
```



Spark Application: Jobs, Stages and Tasks

- On the contrary, narrow operations are performed locally, in a partition basis (with the executor process operating on it without any external communication).

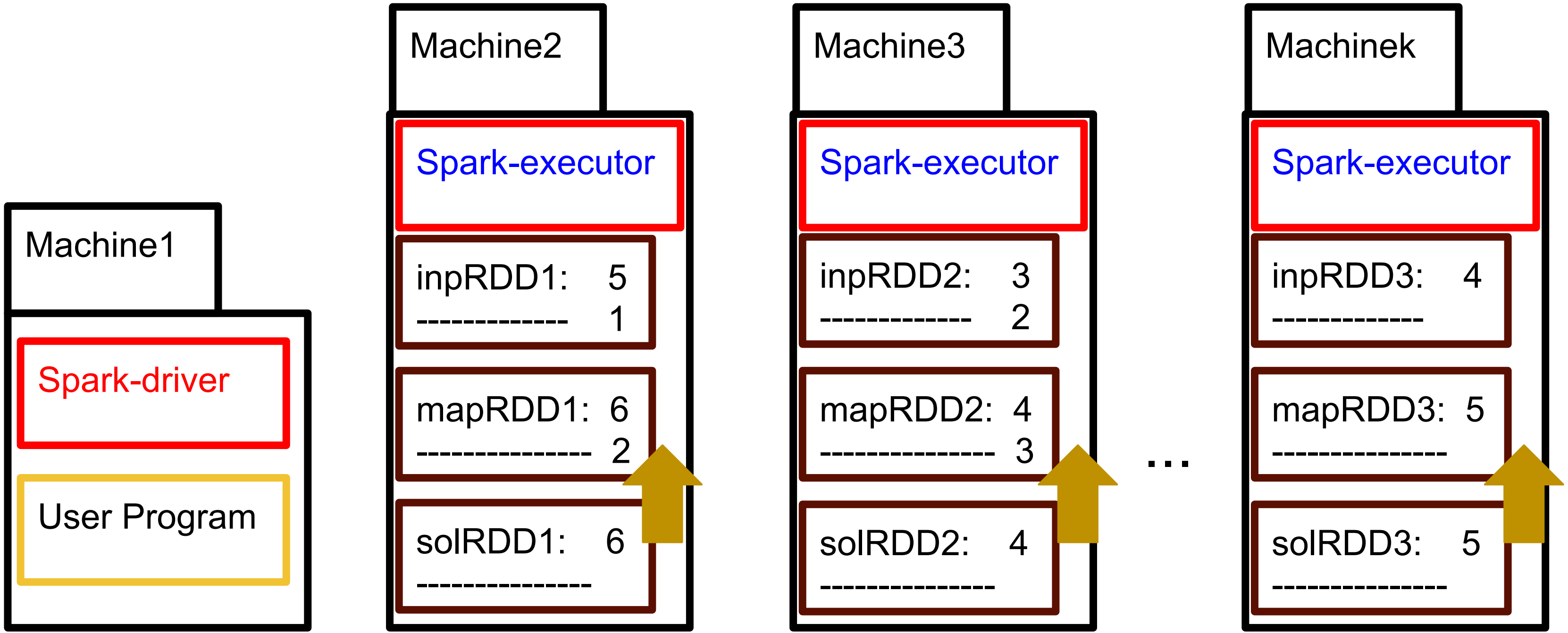
```
inputRDD  = sc.parallelize( [ 1, 2, 3, 4, 5 ] )
mapRDD = inputRDD.map( lambda elem : elem + 1 )
solRDD = mapRDD.filter( lambda elem : elem > 3 )
```



Spark Application: Jobs, Stages and Tasks

- On the contrary, narrow operations are performed locally, in a partition basis (with the executor process operating on it without any external communication).

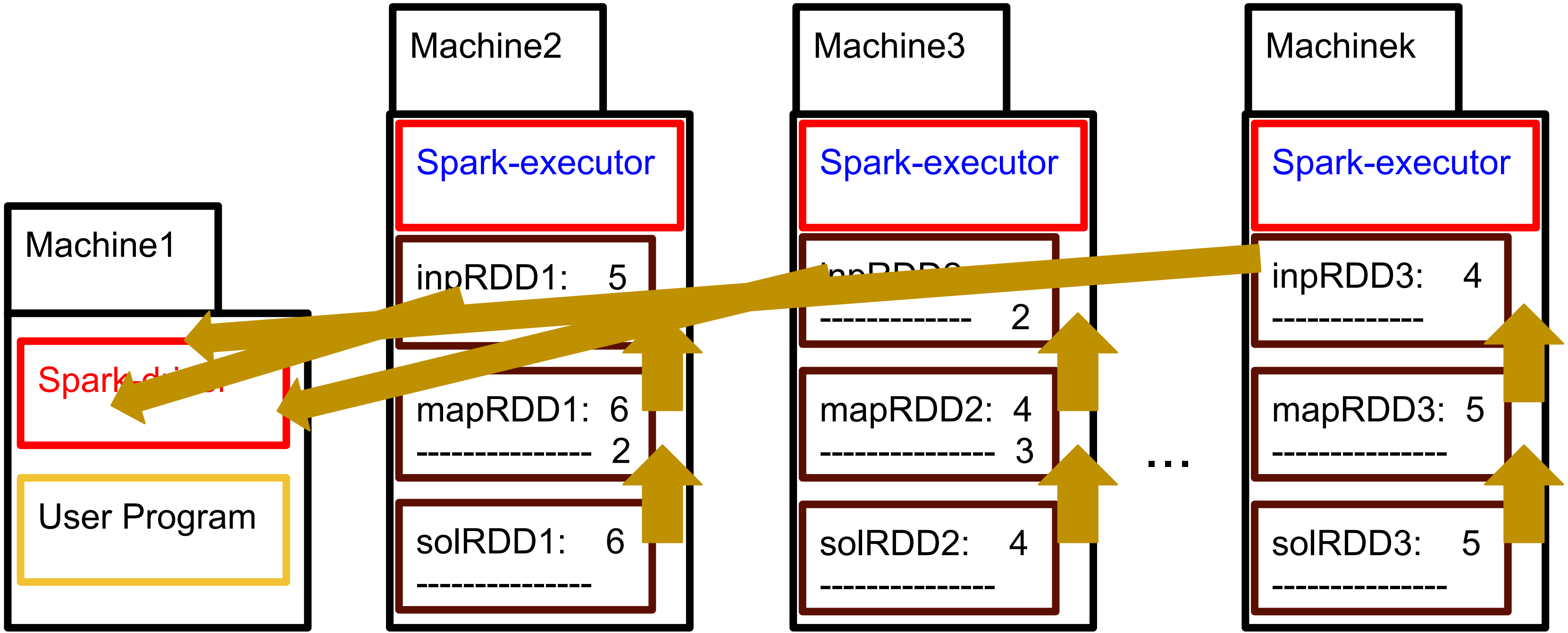
```
inputRDD  = sc.parallelize( [ 1, 2, 3, 4, 5 ] )
mapRDD = inputRDD.map( lambda elem : elem + 1 )
solRDD = mapRDD.filter( lambda elem : elem > 3 )
```



Spark Application: Jobs, Stages and Tasks

- Moreover, these multiple narrow operations can be pipelined to speed-up their computation by processing them all in one go (with just one pass to the partition data).

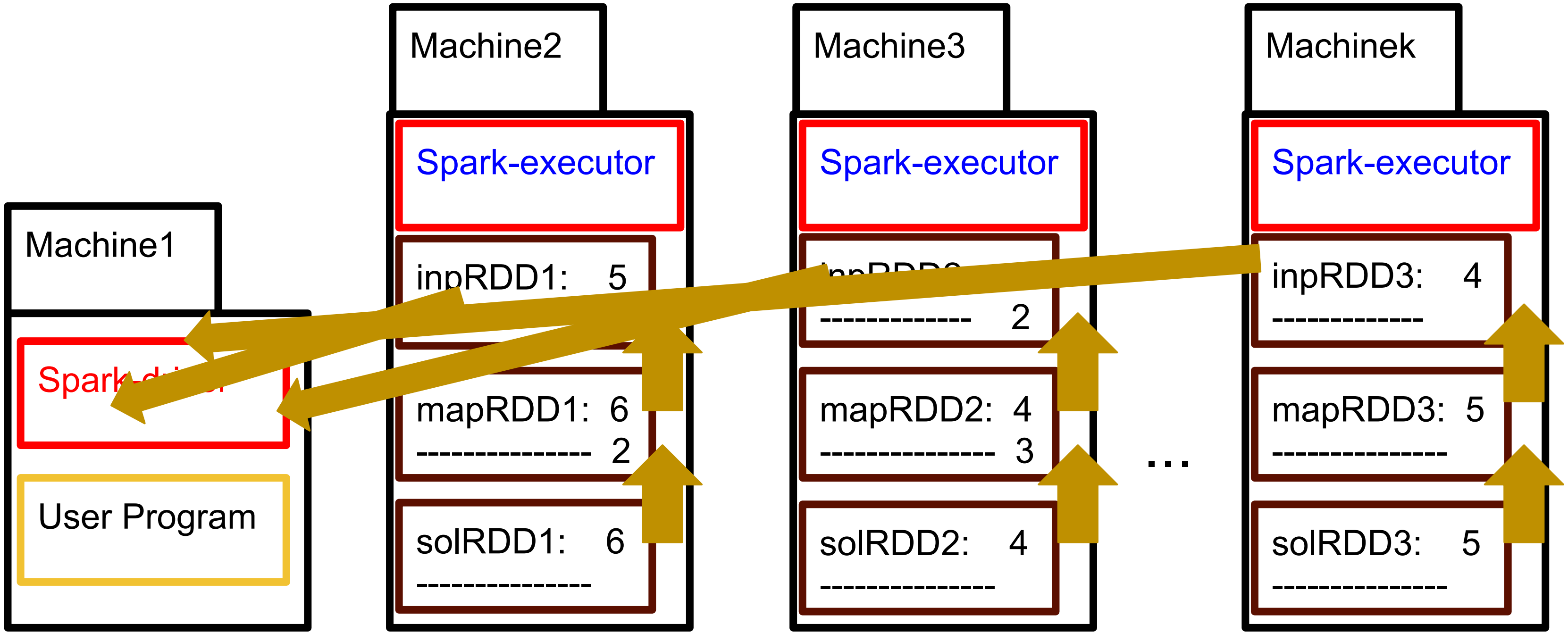
```
inputRDD  = sc.parallelize( [ 1, 2, 3, 4, 5 ] )
mapRDD = inputRDD.map( lambda elem : elem + 1 )
solRDD = mapRDD.filter( lambda elem : elem > 3 )
```



Spark Application: Jobs, Stages and Tasks

- The DAG refers to each of these pipelines of narrow operations as a task.

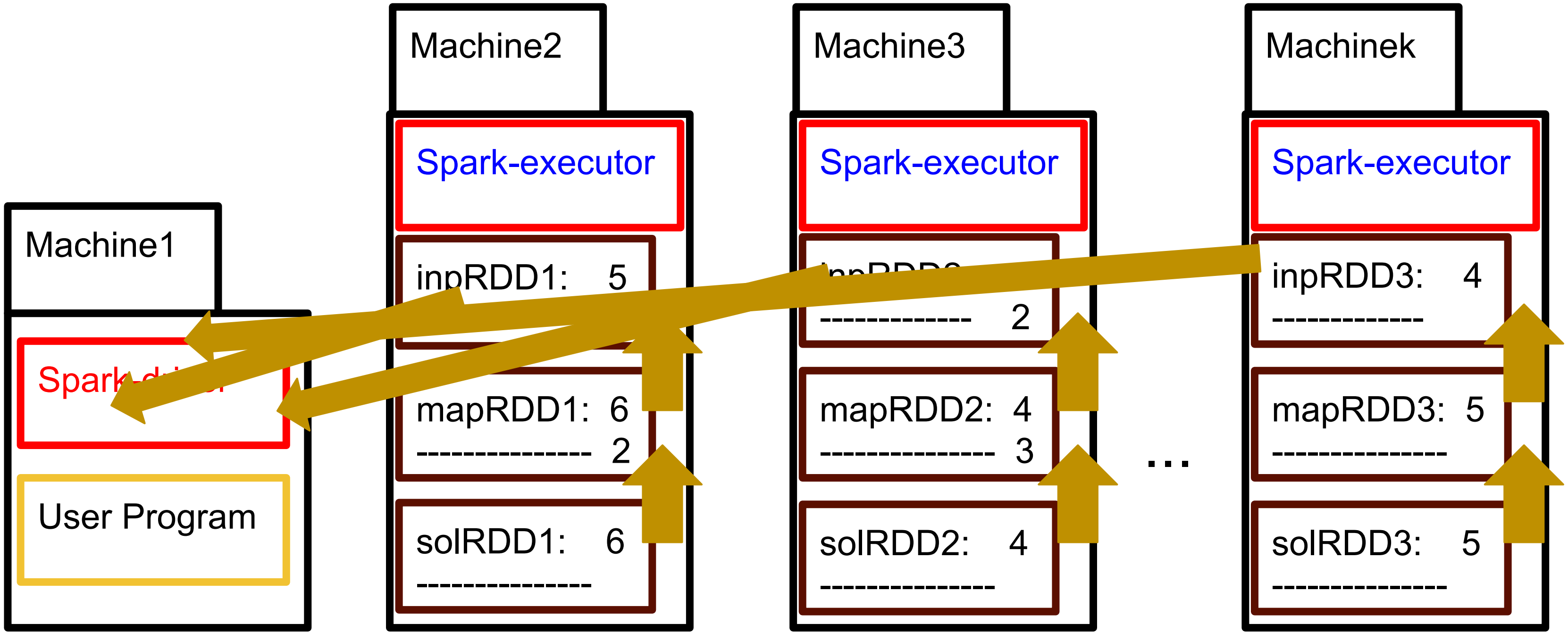
```
inputRDD  = sc.parallelize( [ 1, 2, 3, 4, 5 ] )
mapRDD = inputRDD.map( lambda elem : elem + 1 )
solRDD = mapRDD.filter( lambda elem : elem > 3 )
```



Spark Application: Jobs, Stages and Tasks

- The DAG refers to each of these pipelines of narrow operations as a task. If an RDD is split into **n partitions**, then **n instances of the same task** will be computed in parallel by the Spark executors hosting the n partitions.

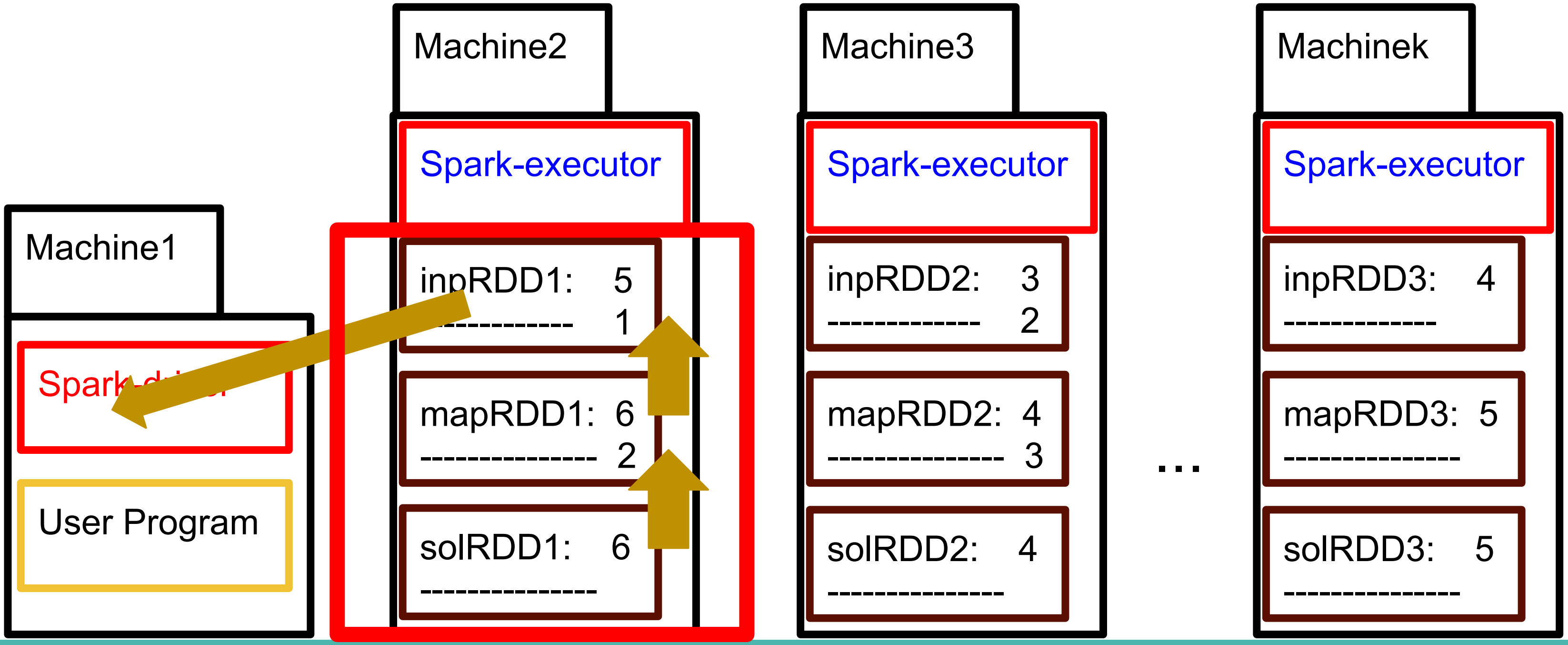
```
inputRDD  = sc.parallelize( [ 1, 2, 3, 4, 5 ] )
mapRDD = inputRDD.map( lambda elem : elem + 1 )
solRDD = mapRDD.filter( lambda elem : elem > 3 )
```



Spark Application: Jobs, Stages and Tasks

- The DAG refers to each of these pipelines of narrow operations as a task.
  For example, in this case we have 3 partitions.
  This is our Task1.

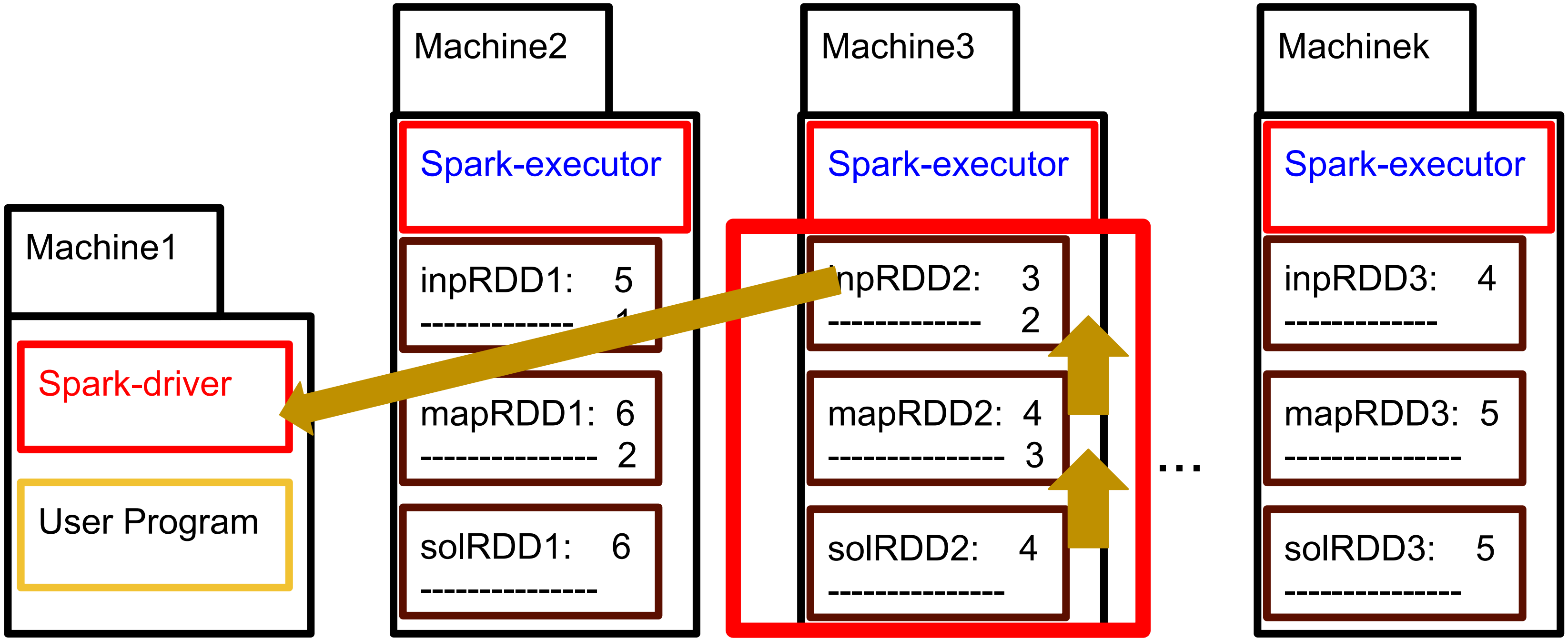
```
inputRDD  = sc.parallelize( [ 1, 2, 3, 4, 5 ] )
mapRDD = inputRDD.map( lambda elem : elem + 1 )
solRDD = mapRDD.filter( lambda elem : elem > 3 )
```



Spark Application: Jobs, Stages and Tasks

- The DAG refers to each of these pipelines of narrow operations as a task.
  For example, in this case we have 3 partitions.
  This is our Task2.

```
inputRDD  = sc.parallelize( [ 1, 2, 3, 4, 5 ] )
mapRDD = inputRDD.map( lambda elem : elem + 1 )
solRDD = mapRDD.filter( lambda elem : elem > 3 )
```



Machine1

Spark-driver

User Program

Machine2

Spark-executor

inpRDD1: 5
1

mapRDD1: 6
2

solRDD1: 6

Machine3

Spark-executor

inpRDD2: 3
2

mapRDD2: 4
3

solRDD2: 4

...

Machinek

Spark-executor

inpRDD3: 4

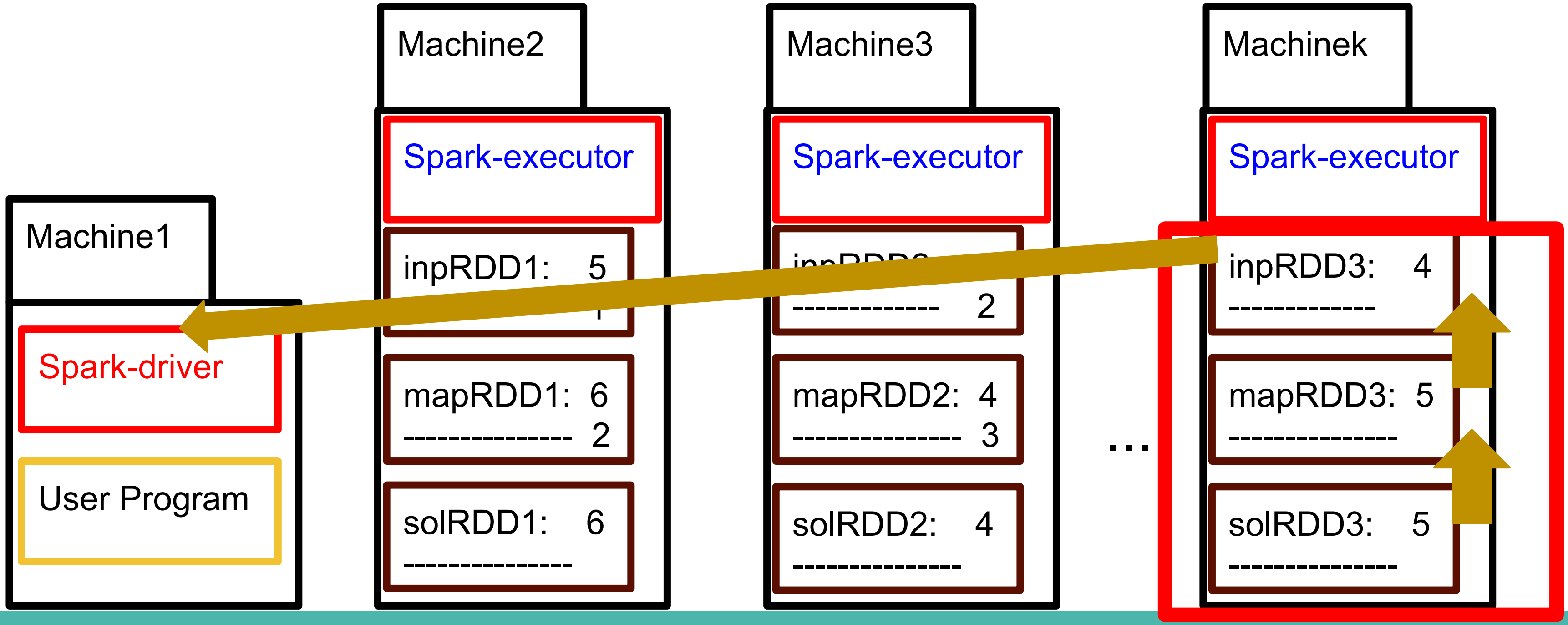
mapRDD3: 5

solRDD3: 5

Spark Application: Jobs, Stages and Tasks

- The DAG refers to each of these pipelines of narrow operations as a task.
  For example, in this case we have 3 partitions.
  This is our Task3.

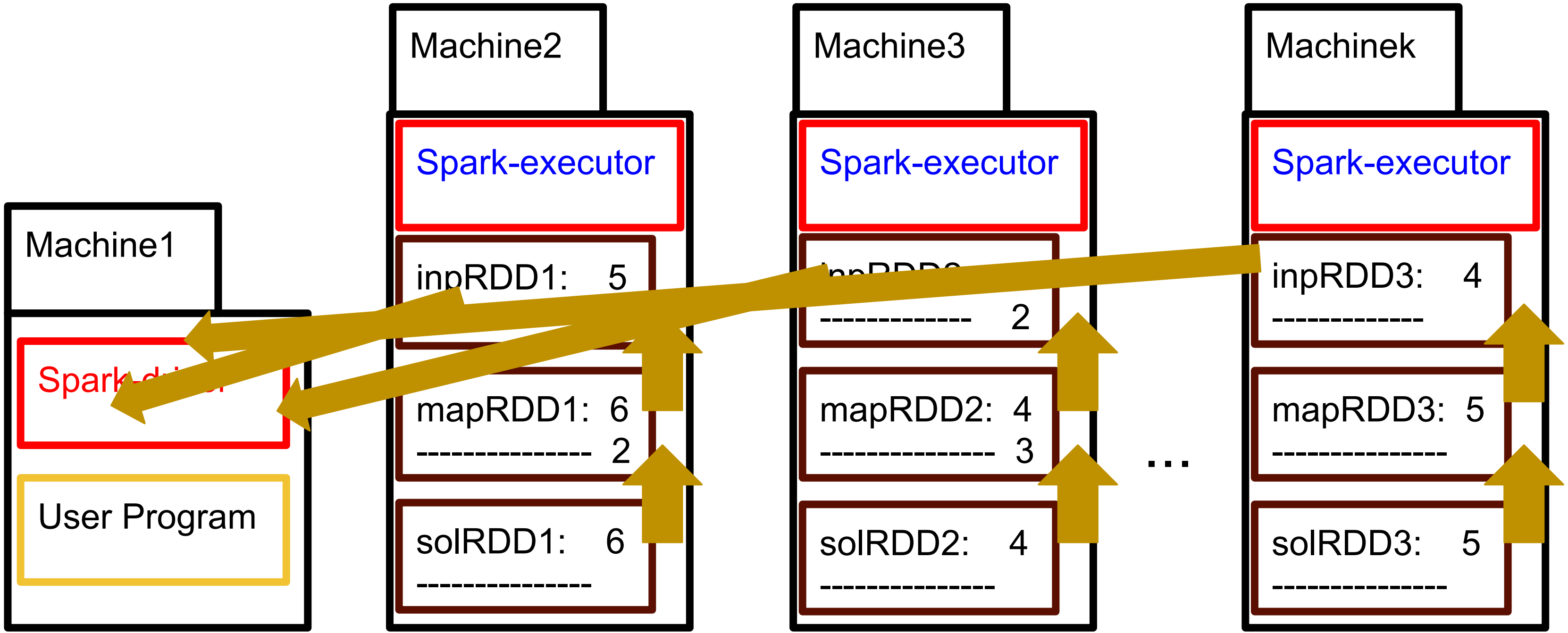
```
inputRDD  = sc.parallelize( [ 1, 2, 3, 4, 5 ] )
mapRDD = inputRDD.map( lambda elem : elem + 1 )
solRDD = mapRDD.filter( lambda elem : elem > 3 )
```



Spark Application: Jobs, Stages and Tasks

- The DAG refers to each of these pipelines of narrow operations as a task. For example, in this case we have 3 partitions. **And again, Task1, Task2 and Task3 can run in parallel.**

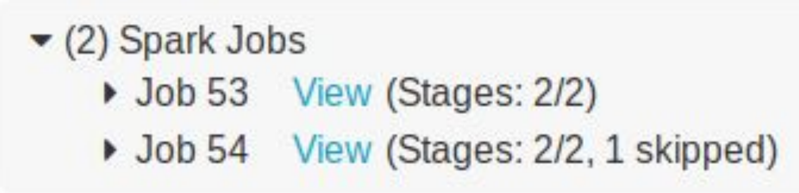
```
inputRDD  = sc.parallelize( [ 1, 2, 3, 4, 5 ] )
mapRDD = inputRDD.map( lambda elem : elem + 1 )
solRDD = mapRDD.filter( lambda elem : elem > 3 )
```



Spark Application: Jobs, Stages and Tasks

- Back to our program examples with Jobs 53 and 54:

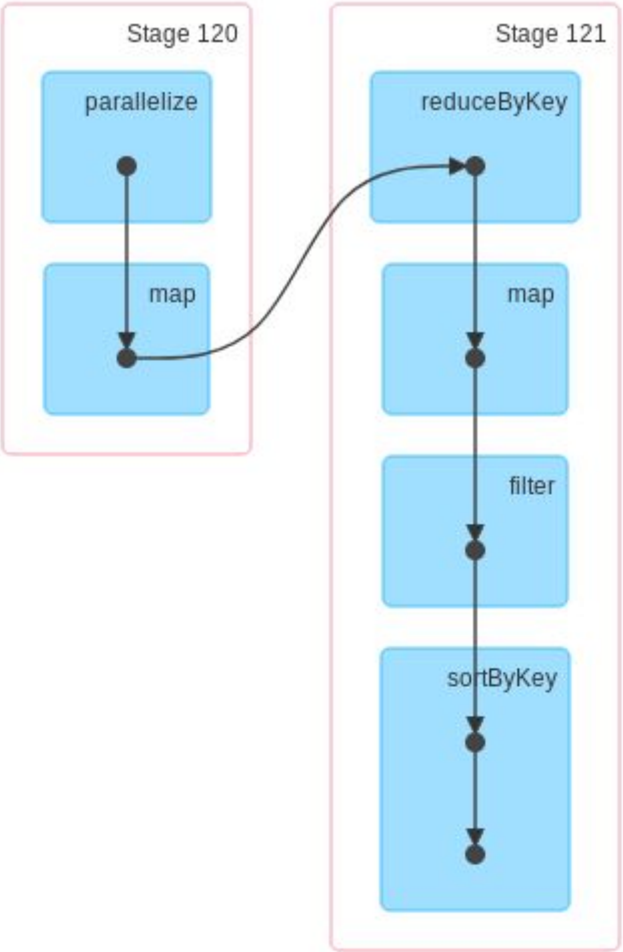
```
inputRDD  = sc.parallelize( [ "Hello", "Hola", "Bonjour","Hello",
                                  "Bonjour", "Ciao", "Hello" ] )
pairWordsRDD = inputRDD.map( lambda x : (x, 1) )
countRDD = pairWordsRDD.reduceByKey( lambda x, y : x + y )
swapTupleRDD = countRDD.map( lambda x : (x[1], x[0]) )
filteredRDD = swapTupleRDD.filter( lambda x : x[0] > 1 )
solutionRDD = filteredRDD.sortByKey()
resVAL = solutionRDD.collect()
```



▾ (2) Spark Jobs
- ▸ Job 53 View (Stages: 2/2)
- ▸ Job 54 View (Stages: 2/2, 1 skipped)

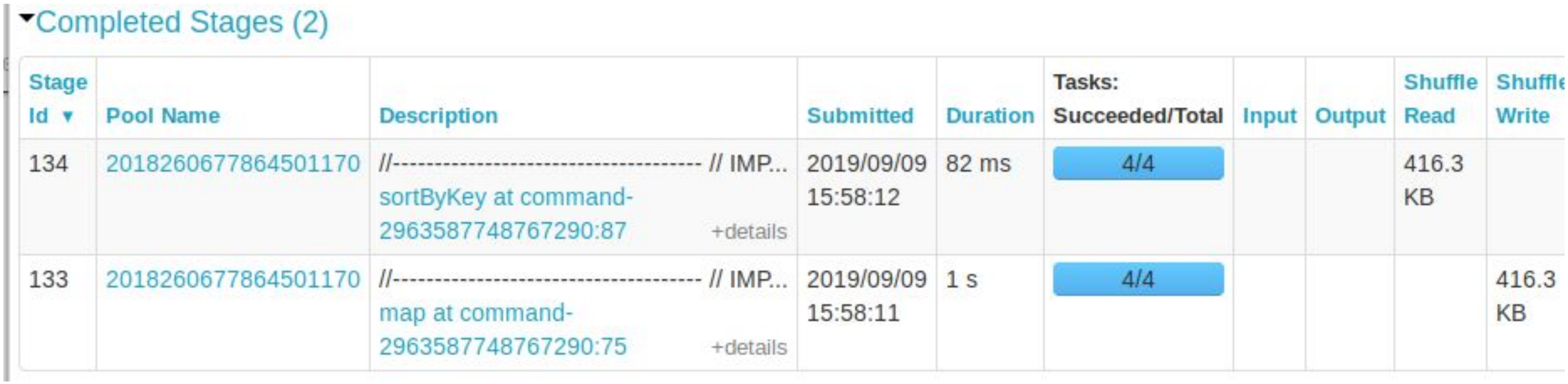
Spark Application: Jobs, Stages and Tasks

- This was Job 53.



Spark Application: Jobs, Stages and Tasks

- And it leads to the following tasks:



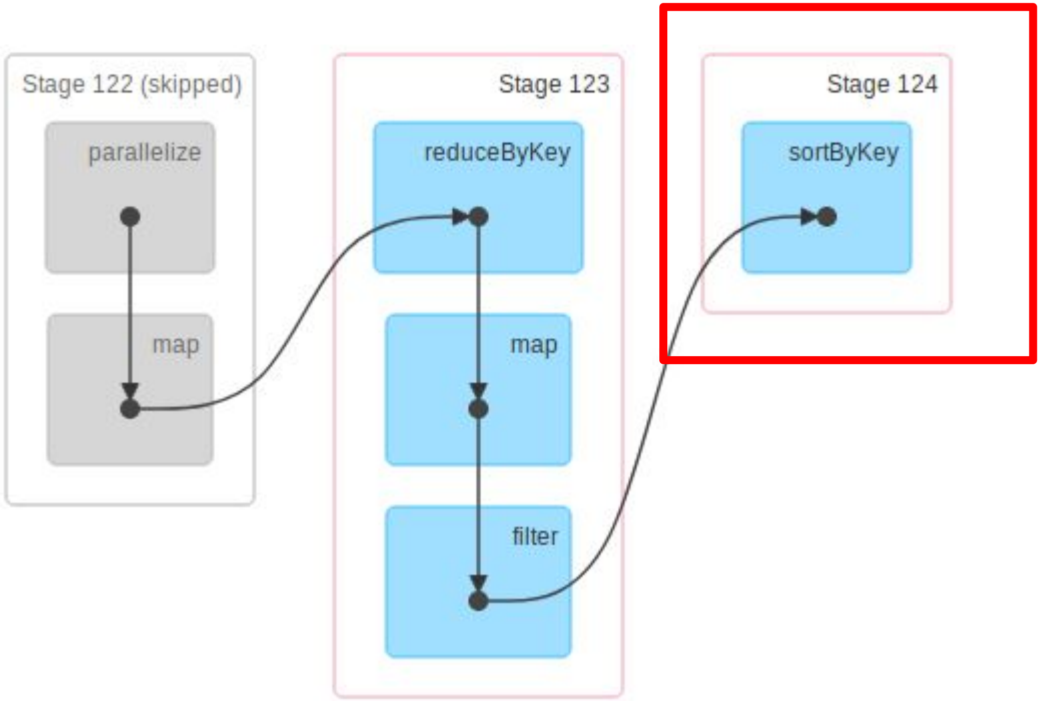
▾Completed Stages (2)

Stage Id ▾	Pool Name	Description	Submitted	Duration	Tasks: Succeeded/Total	Input	Output	Shuffle Read	Shuffle Write
134	2018260677864501170	//------------------------------------ // IMP... sortByKey at command-2963587748767290:87 +details	2019/09/09 15:58:12	82 ms	4/4			416.3 KB	
133	2018260677864501170	//------------------------------------ // IMP... map at command-2963587748767290:75 +details	2019/09/09 15:58:11	1 s	4/4				416.3 KB

The different numbers in the Stage id are because I re-run the examples

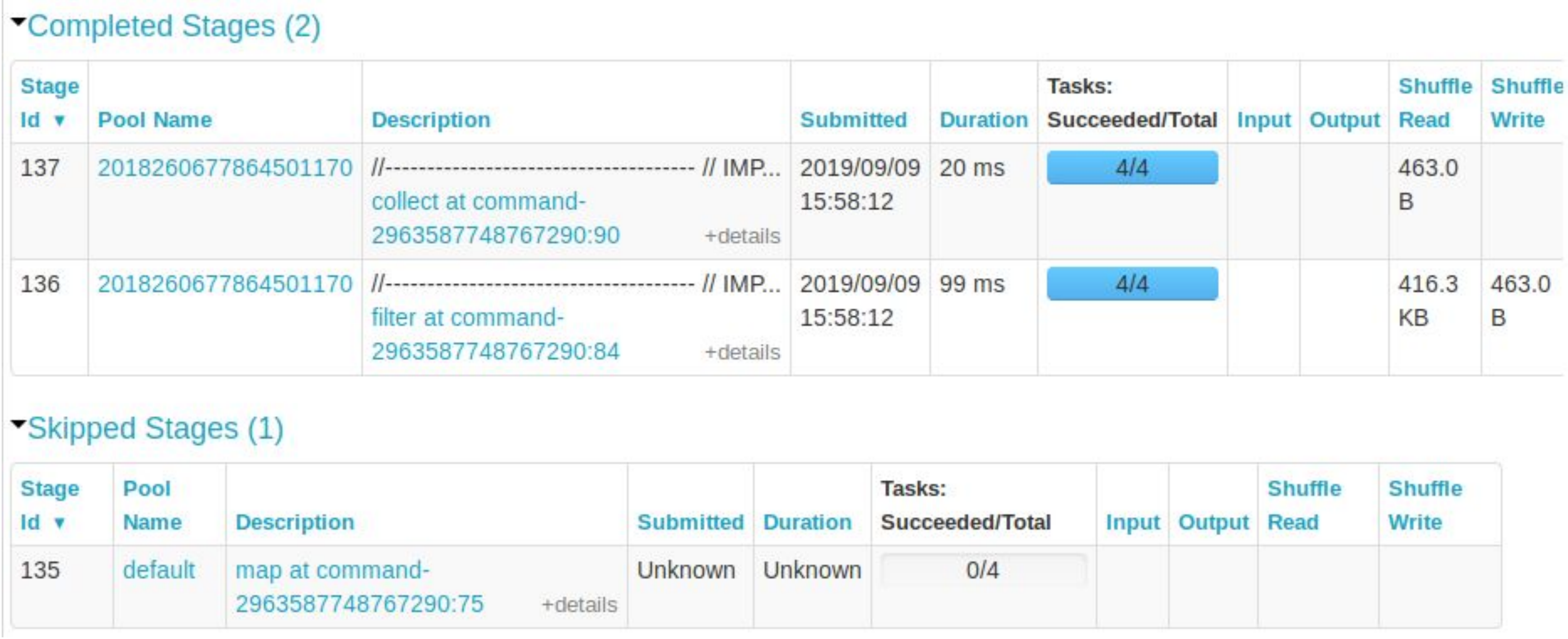
Spark Application: Jobs, Stages and Tasks

- This was Job 54.



Spark Application: Jobs, Stages and Tasks

- And it leads to the following tasks:



▾Completed Stages (2)

Stage Id ▾	Pool Name	Description	Submitted	Duration	Tasks: Succeeded/Total	Input	Output	Shuffle Read	Shuffle Write
137	2018260677864501170	//------------------------------------ // IMP... collect at command-2963587748767290:90 +details	2019/09/09 15:58:12	20 ms	4/4			463.0 B	
136	2018260677864501170	//------------------------------------ // IMP... filter at command-2963587748767290:84 +details	2019/09/09 15:58:12	99 ms	4/4			416.3 KB	463.0 B

▾Skipped Stages (1)

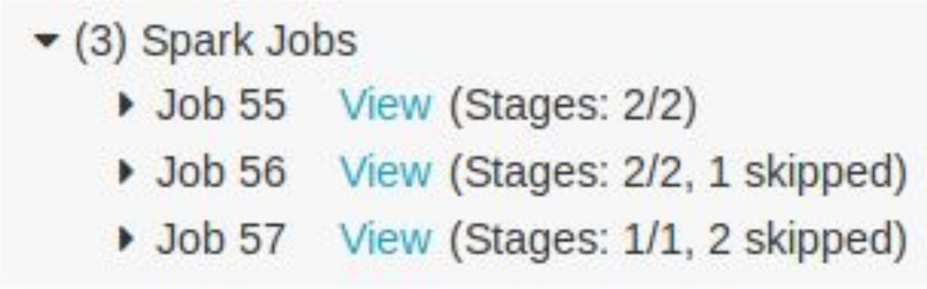
Stage Id ▾	Pool Name	Description	Submitted	Duration	Tasks: Succeeded/Total	Input	Output	Shuffle Read	Shuffle Write
135	default	map at command-2963587748767290:75 +details	Unknown	Unknown	0/4				

The different numbers in the Stage id are because I re-run the examples

Spark Application: Jobs, Stages and Tasks

- Back to our program examples with Jobs 55, 56 and 57:

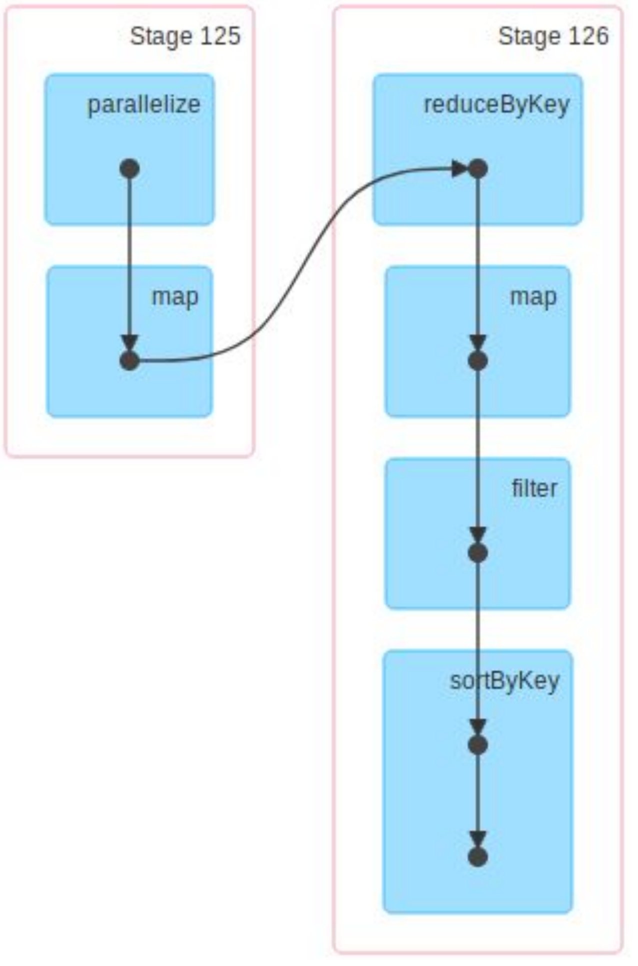
```
inputRDD  = sc.parallelize( [ "Hello", "Hola", "Bonjour","Hello",
                                  "Bonjour", "Ciao", "Hello" ] )
pairWordsRDD = inputRDD.map( lambda x : (x, 1) )
countRDD = pairWordsRDD.reduceByKey( lambda x, y : x + y )
swapTupleRDD = countRDD.map( lambda x : (x[1], x[0]) )
filteredRDD = swapTupleRDD.filter( lambda x : x[0] > 1 )
solutionRDD = filteredRDD.sortByKey()
solutionRDD.persist()
resVAL1 = solutionRDD.collect()
resVAL2 = solutionRDD.count()
```



```
▾ (3) Spark Jobs
    ▸ Job 55   View (Stages: 2/2)
    ▸ Job 56   View (Stages: 2/2, 1 skipped)
    ▸ Job 57   View (Stages: 1/1, 2 skipped)
```

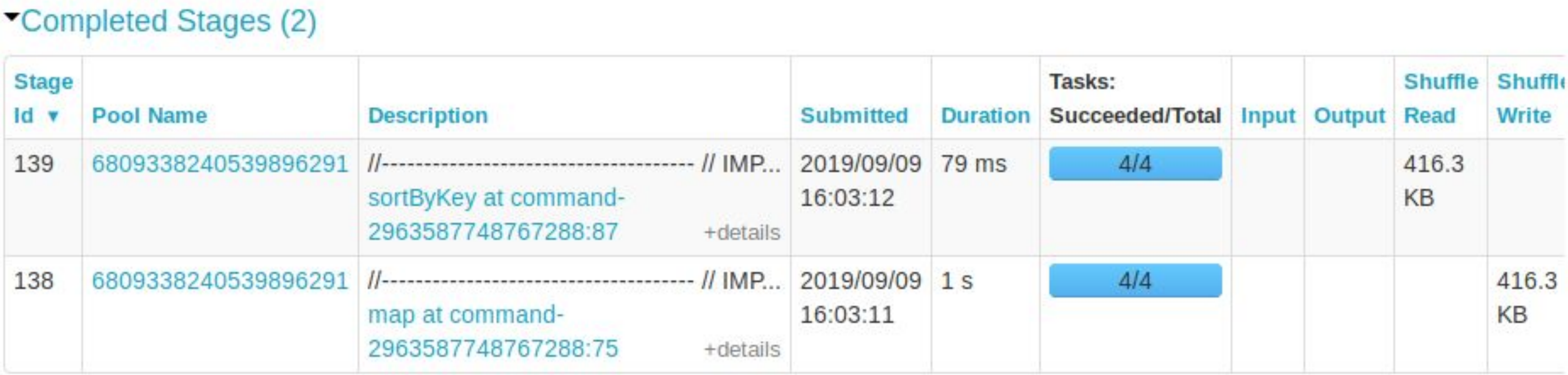
Spark Application: Jobs, Stages and Tasks

- This was Job 55.



Spark Application: Jobs, Stages and Tasks

- And it leads to the following tasks:



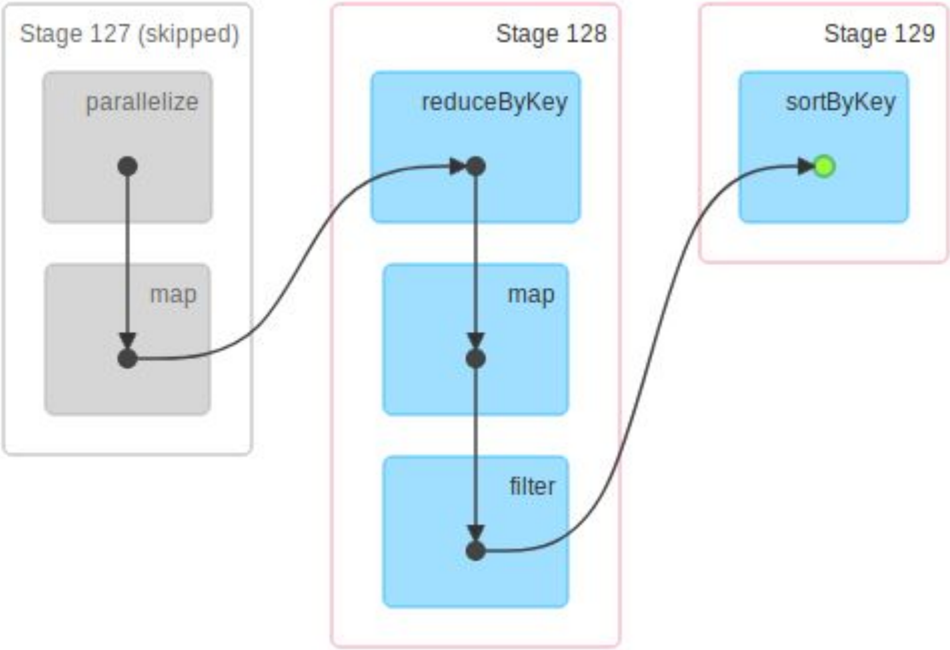
Completed Stages (2)

Stage Id ▾	Pool Name	Description	Submitted	Duration	Tasks: Succeeded/Total	Input	Output	Shuffle Read	Shuffl Write
139	6809338240539896291	//-------------------------------------- // IMP... sortByKey at command-2963587748767288:87 +details	2019/09/09 16:03:12	79 ms	4/4			416.3 KB	
138	6809338240539896291	//-------------------------------------- // IMP... map at command-2963587748767288:75 +details	2019/09/09 16:03:11	1 s	4/4				416.3 KB

The different numbers in the Stage id are because I re-run the examples

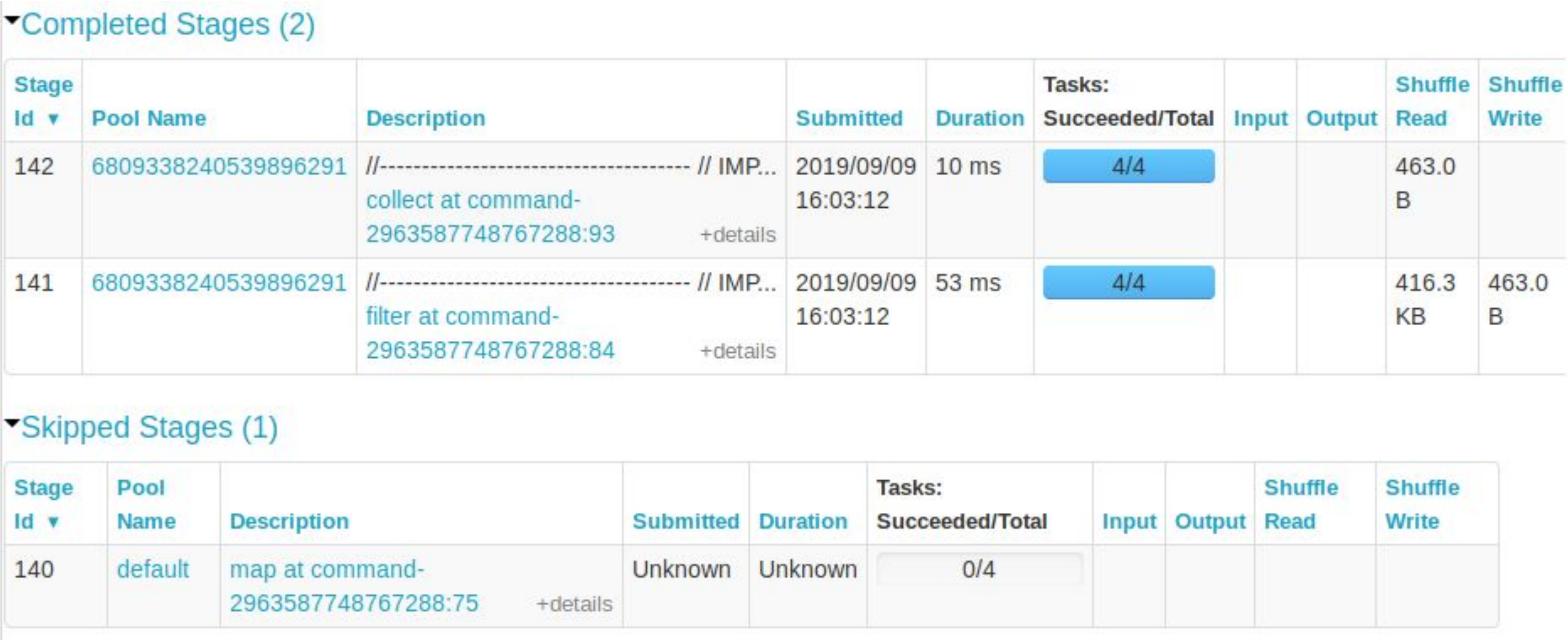
Spark Application: Jobs, Stages and Tasks

- This was Job 56.



Spark Application: Jobs, Stages and Tasks

- And it leads to the following tasks:



Completed Stages (2)

Stage Id ▾	Pool Name	Description	Submitted	Duration	Tasks: Succeeded/Total	Input	Output	Shuffle Read	Shuffle Write
142	6809338240539896291	//-------------------------------------- // IMP... collect at command-2963587748767288:93 +details	2019/09/09 16:03:12	10 ms	4/4			463.0 B	
141	6809338240539896291	//-------------------------------------- // IMP... filter at command-2963587748767288:84 +details	2019/09/09 16:03:12	53 ms	4/4			416.3 KB	463.0 B

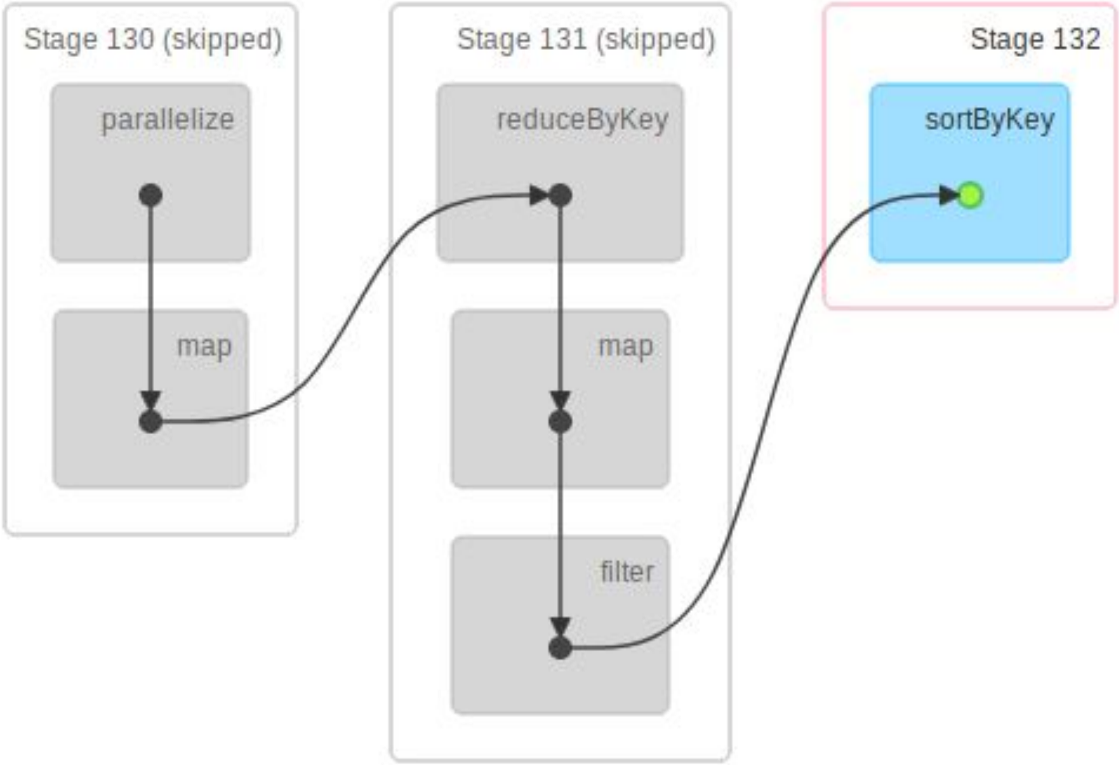
Skipped Stages (1)

Stage Id ▾	Pool Name	Description	Submitted	Duration	Tasks: Succeeded/Total	Input	Output	Shuffle Read	Shuffle Write
140	default	map at command-2963587748767288:75 +details	Unknown	Unknown	0/4				

The different numbers in the Stage id are because I re-run the examples

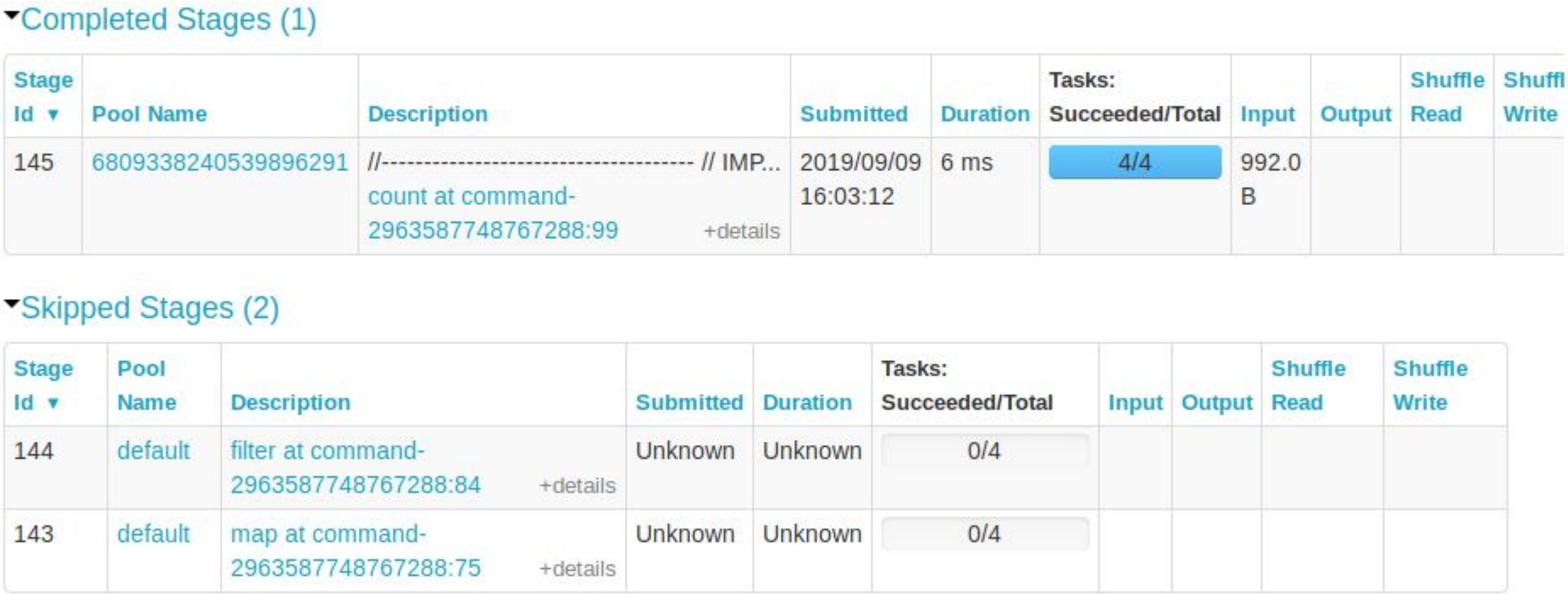
Spark Application: Jobs, Stages and Tasks

- This was Job 57.



Spark Application: Jobs, Stages and Tasks

- And it leads to the following tasks:



▾Completed Stages (1)

Stage Id ▾	Pool Name	Description	Submitted	Duration	Tasks: Succeeded/Total	Input	Output	Shuffle Read	Shuffl Write
145	6809338240539896291	//-- // IMP... count at command-2963587748767288:99 +details	2019/09/09 16:03:12	6 ms	4/4	992.0 B			

▾Skipped Stages (2)

Stage Id ▾	Pool Name	Description	Submitted	Duration	Tasks: Succeeded/Total	Input	Output	Shuffle Read	Shuffle Write
144	default	filter at command-2963587748767288:84 +details	Unknown	Unknown	0/4				
143	default	map at command-2963587748767288:75 +details	Unknown	Unknown	0/4				

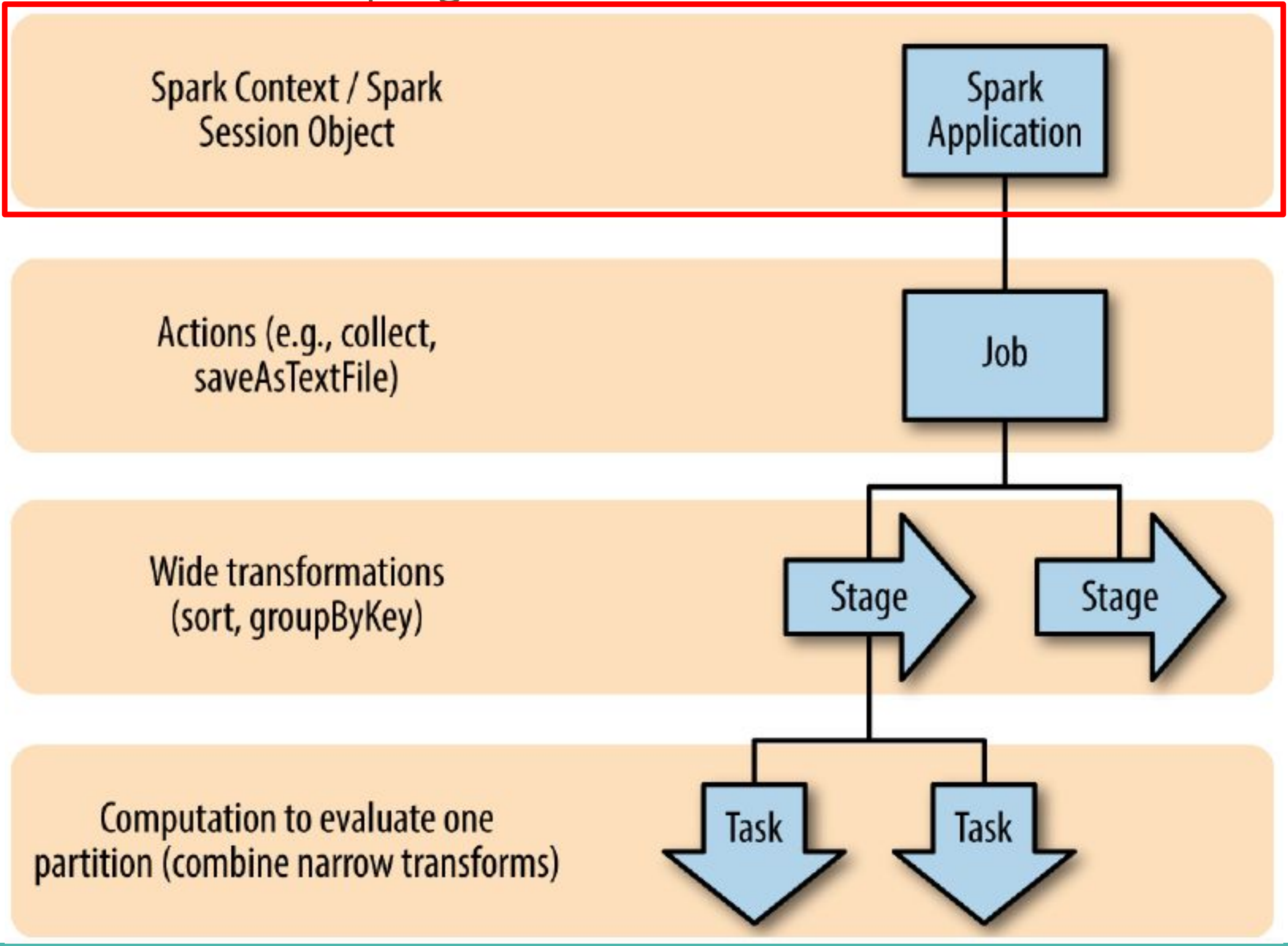
The different numbers in the Stage id are because I re-run the examples

Spark Application: Jobs, Stages and Tasks

So, all in all…

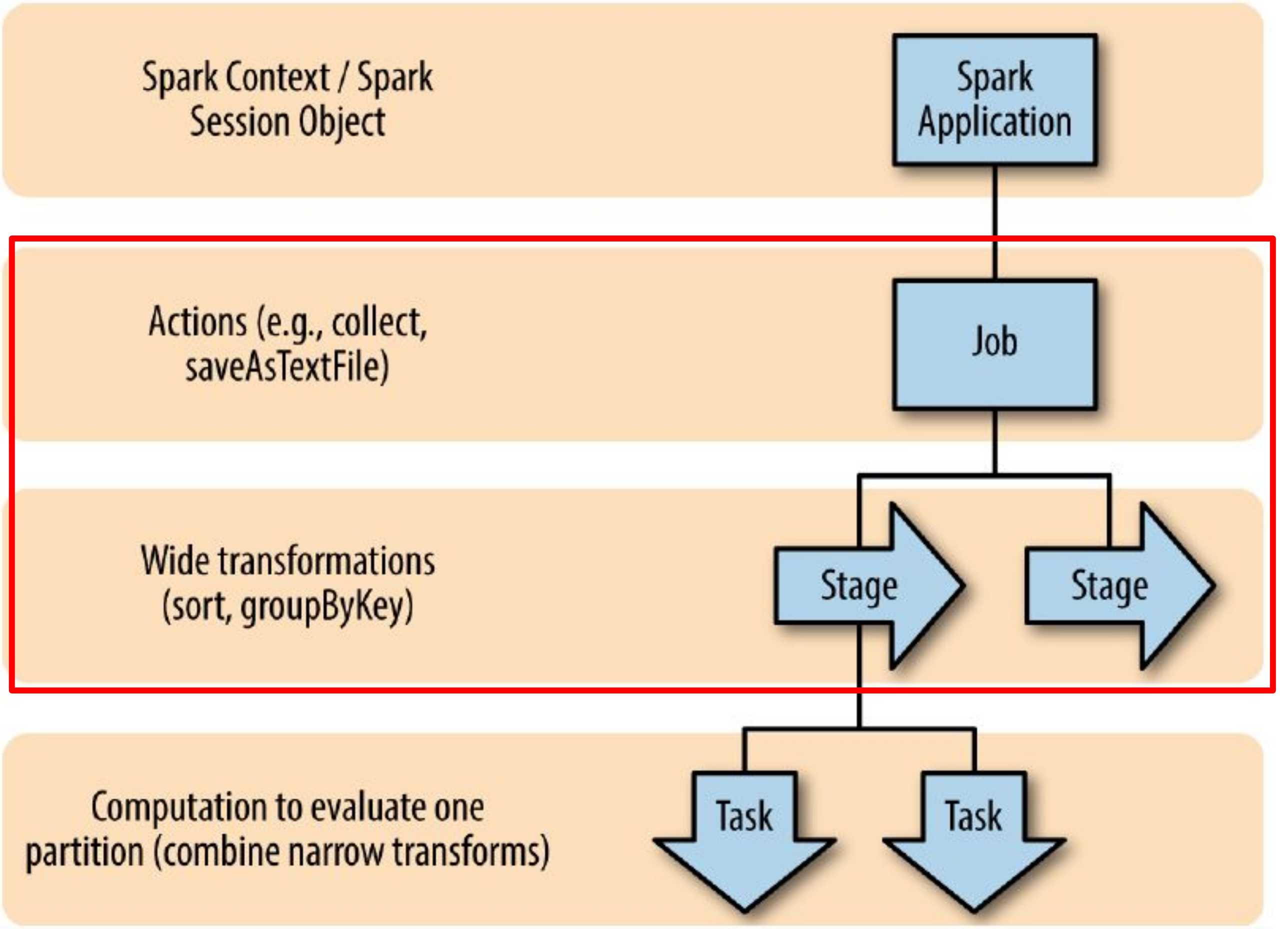
Spark Application: Jobs, Stages and Tasks

- We define a Spark application as a set of Jobs triggered by the **action** operations of the user program.


Spark Context / Spark Session Object
Spark Application
Actions (e.g., collect, saveAsTextFile)
Job
Wide transformations (sort, groupByKey)
Stage
Stage
Computation to evaluate one partition (combine narrow transforms)
Task
Task

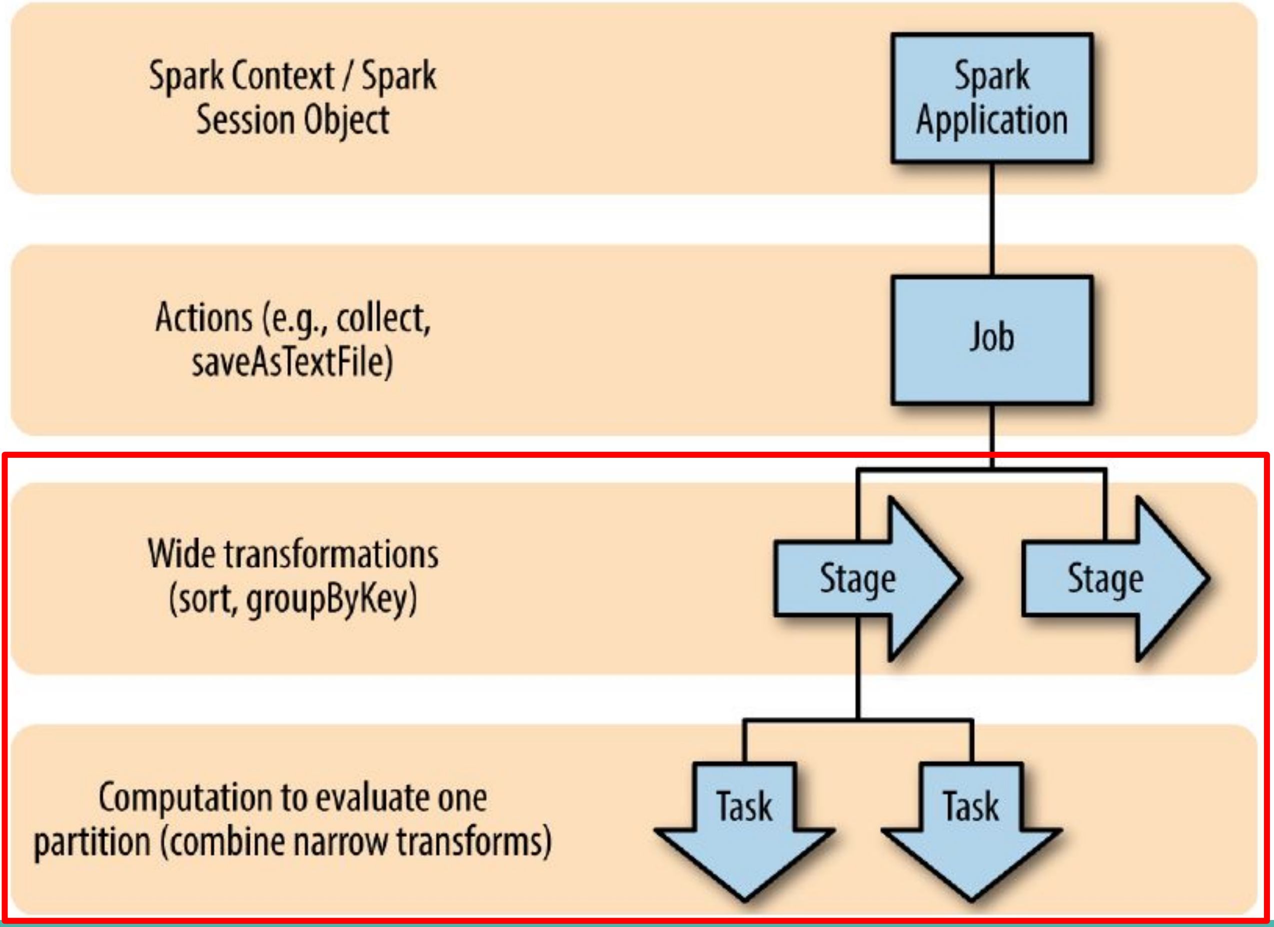
Spark Application: Jobs, Stages and Tasks

- We define a single Job as the sequential execution of stages.
  Each new stage is caused by a wide operation.
  As data is shuffled among stages they must be executed sequentially.


Spark Context / Spark Session Object
Spark Application
Actions (e.g., collect, saveAsTextFile)
Job
Wide transformations (sort, groupByKey)
Stage
Stage
Computation to evaluate one partition (combine narrow transforms)
Task
Task

Spark Application: Jobs, Stages and Tasks

- We define a Stage as the parallel execution of Tasks.
  Each task is a pipeline of narrow operations performed in one go by a single **executor process** on a single partition.


Spark Context / Spark Session Object
Spark Application
Actions (e.g., collect, saveAsTextFile)
Job
Wide transformations (sort, groupByKey)
Stage
Stage
Computation to evaluate one partition (combine narrow transforms)
Task
Task

Outline

1. Setting the Context.
2. RDD Private Side: Partitions and Lineage.
3. Spark Application: Jobs, Stages and Tasks.

Thank you for your attention!