Execution Plans, Lazy Evaluation, and Caching

DSC 232R

1.2 Lazy Evaluation

- **Postpone** computing the square until result is needed.
- No need to store intermediate results.
- Scan through the data once, rather than twice.

1.1 Task: calculate the sum of squares

$$\sum_{i=1}^{n} {x^2}_i$$

The standard (or **busy**) way to do this is

1. Calculate the square of each element.
2. Sum the squares.

This requires **storing** all intermediate results.

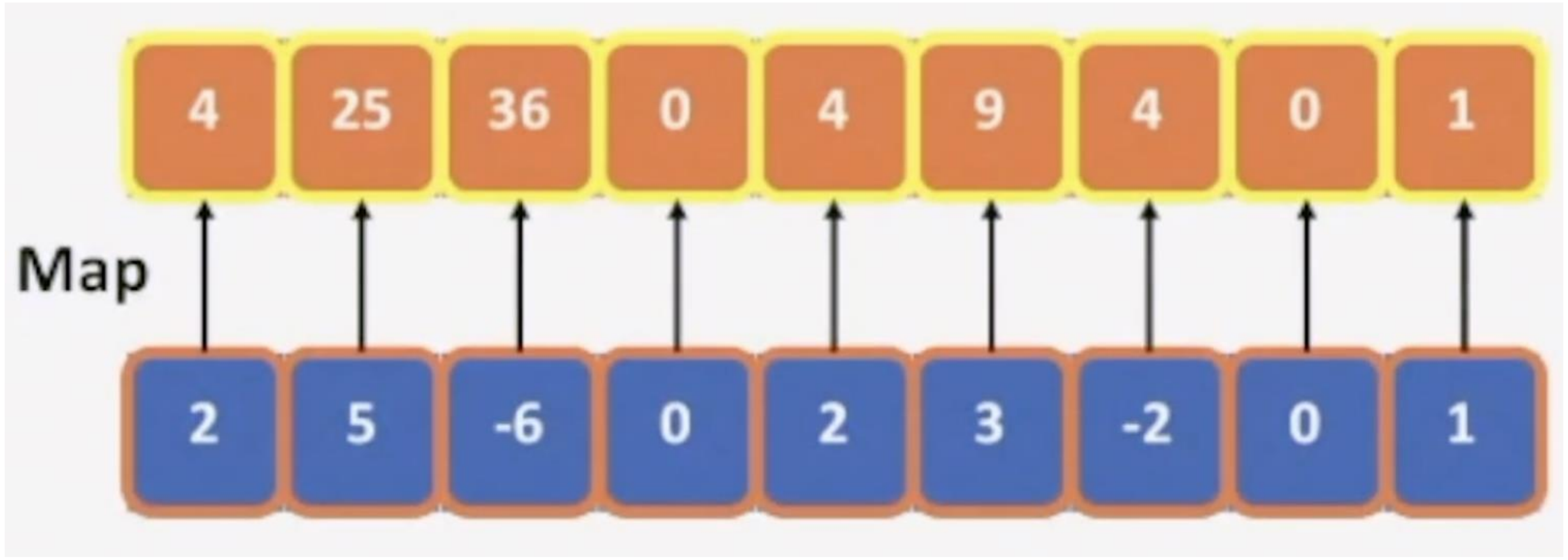
Busy Evaluation

$$S = \sum_{i=1}^{n} x^2{}_i$$

2	5	-6	0	2	3	-2	0	1

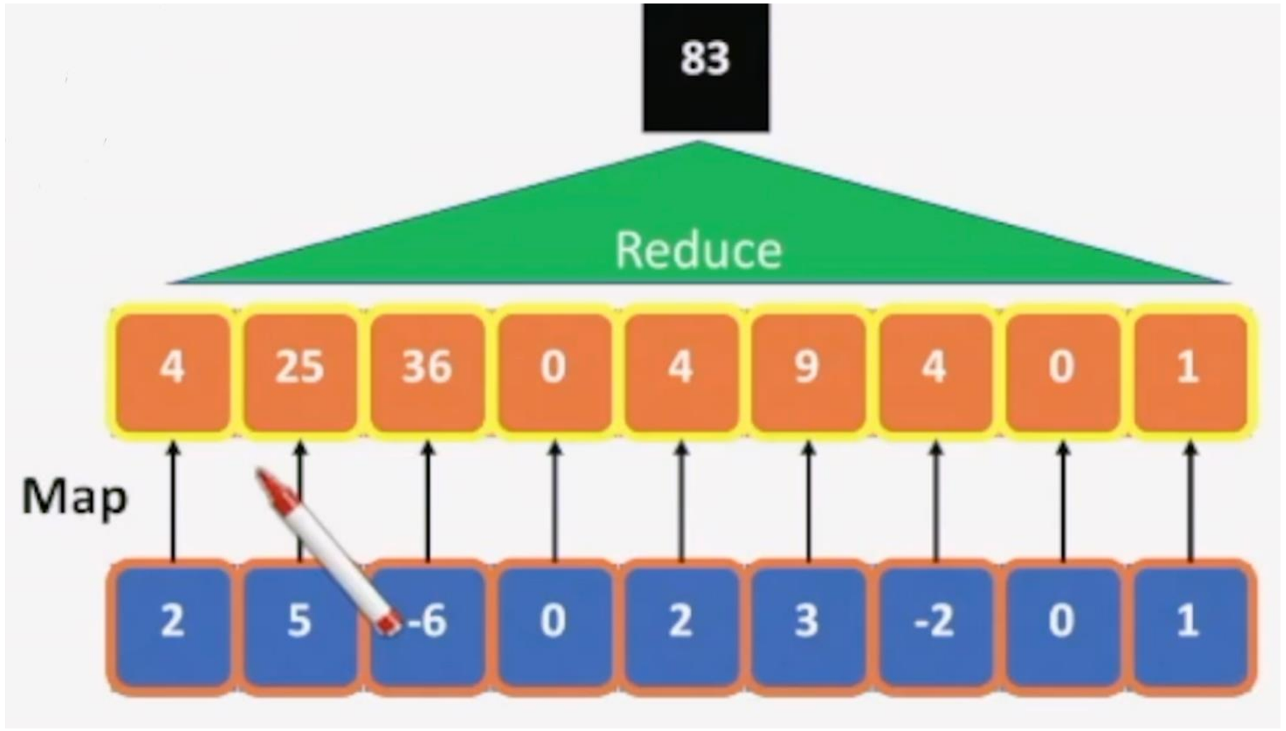
Busy Evaluation

$$S = \sum_{i=1}^{n} x^2{}_i$$



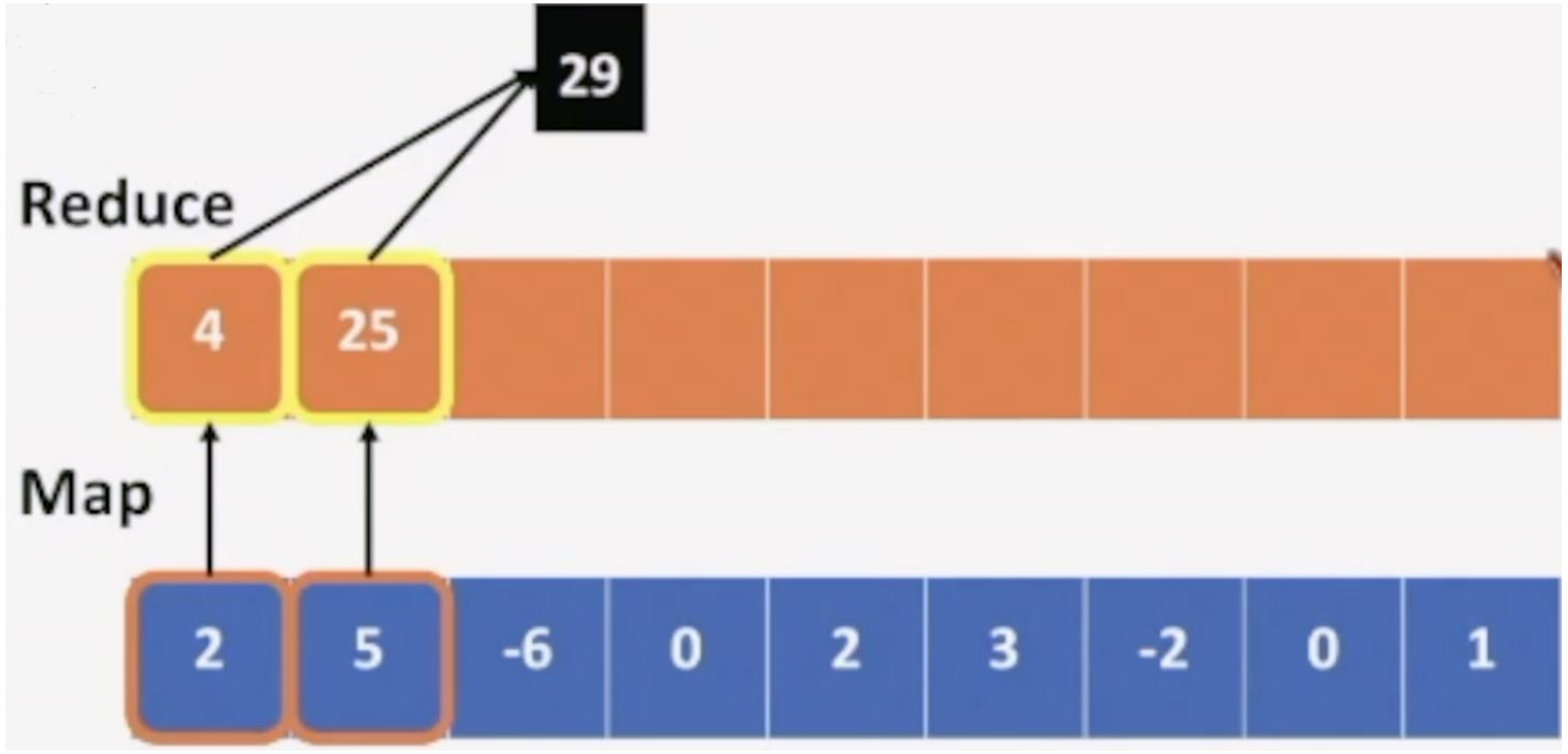
Busy Evaluation

$$S = \sum_{i=1}^{n} x^2{}_i$$



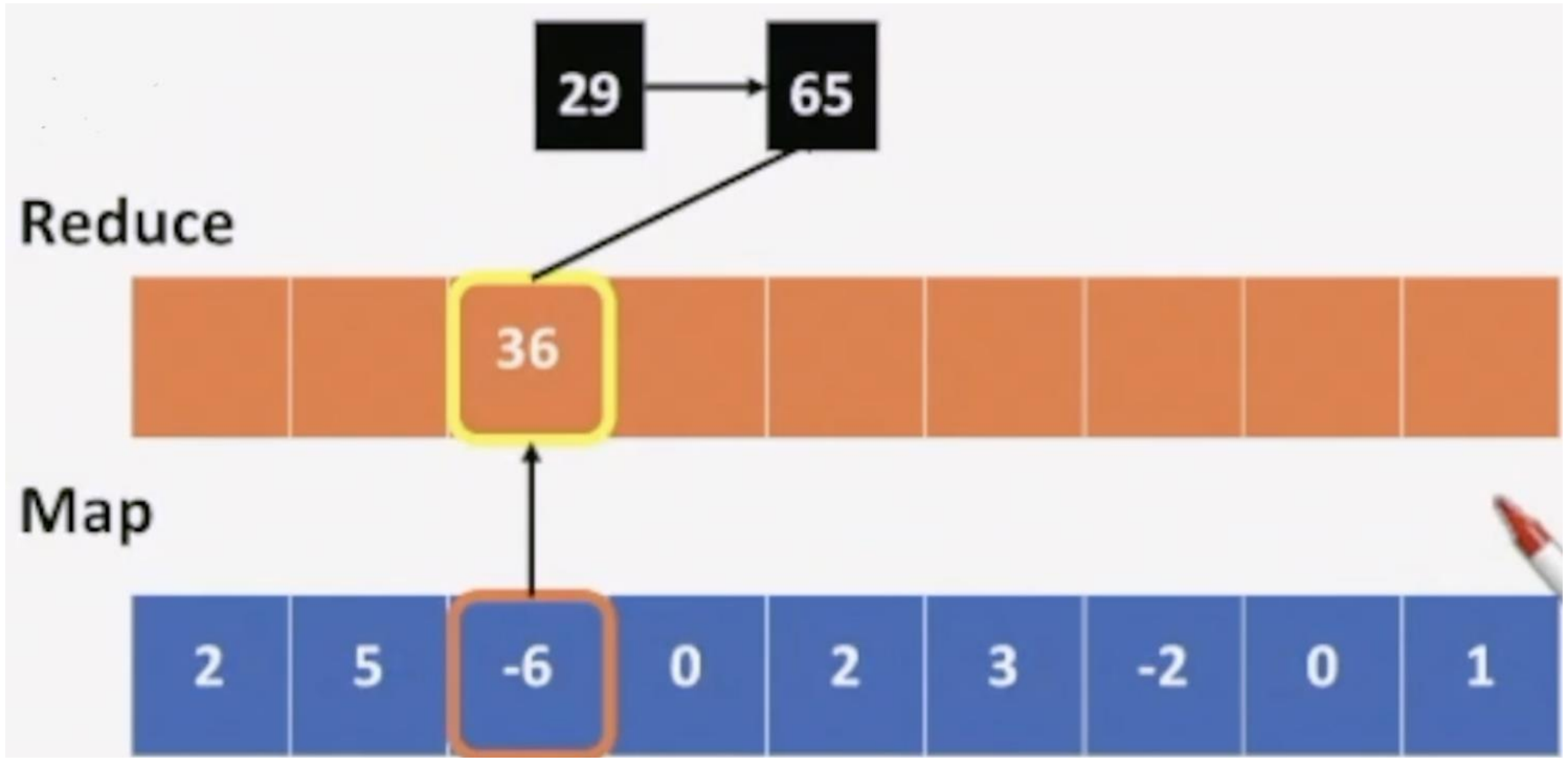
Lazy Evaluation

$$S = \sum_{i=1}^{n} {x^2}_i$$



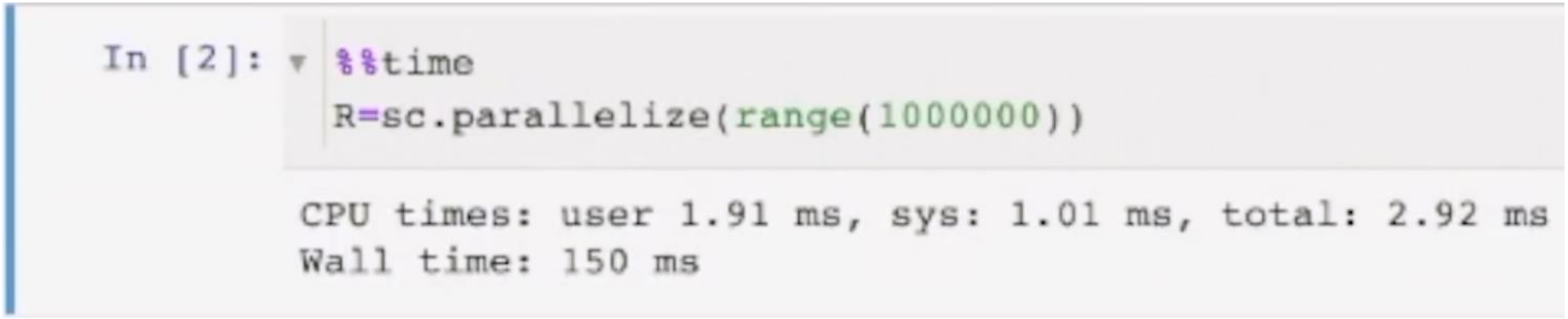
Lazy Evaluation

$$S = \sum_{i=1}^{n} x^2{}_i$$



2 Experimenting with Lazy Evaluation

We create an RDD with one million elements to demonstrate the effects of lazy evaluation.

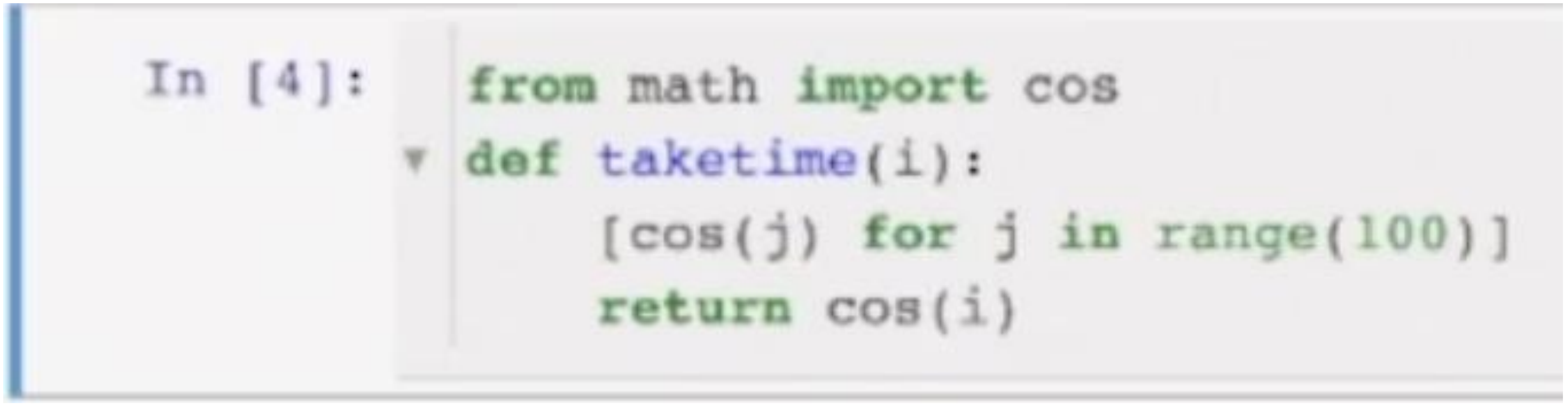


```
In [2]:  %%time
         R=sc.parallelize(range(1000000))
```

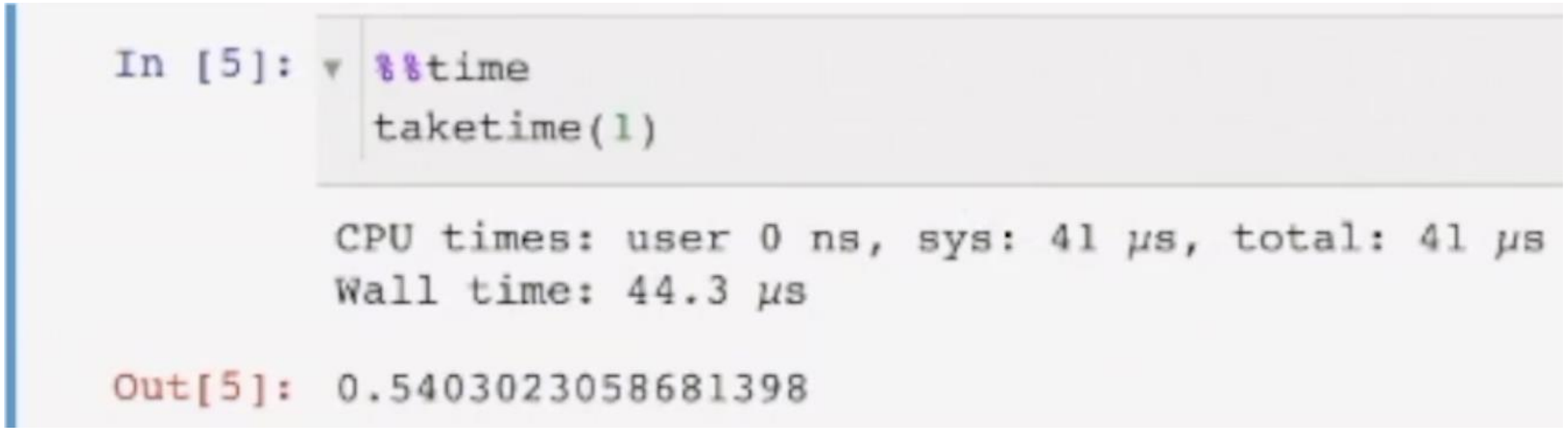
```
CPU times: user 1.91 ms, sys: 1.01 ms, total: 2.92 ms
Wall time: 150 ms
```

2.2 Define a Computation

The role of the function `taketime` is to consume CPU cycles.



```
In [4]:  from math import cos
         def taketime(i):
             [cos(j) for j in range(100)]
             return cos(i)
```



```
In [5]:  %%time
         taketime(1)

CPU times: user 0 ns, sys: 41 µs, total: 41 µs
Wall time: 44.3 µs

Out[5]:  0.5403023058681398
```

2.3 Time Units

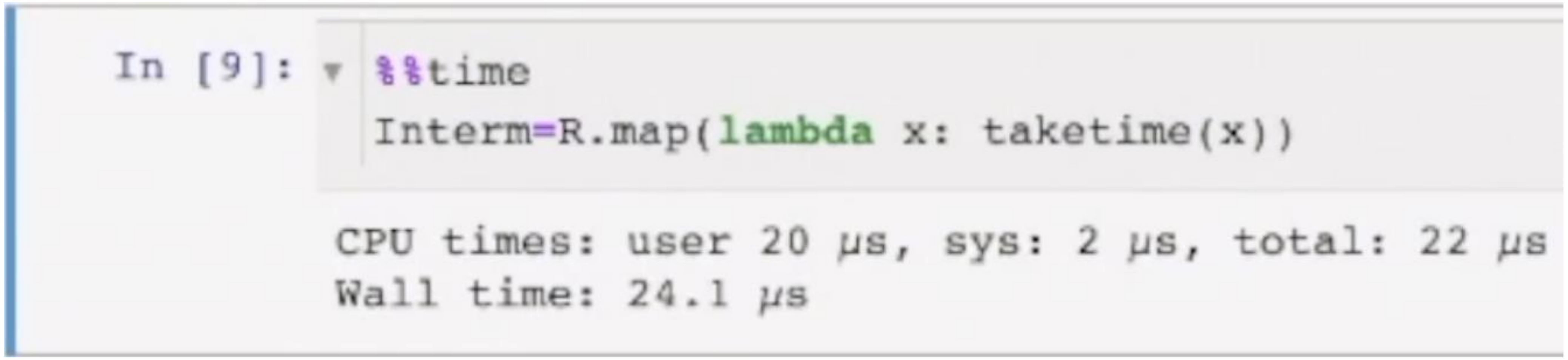
- 1 second = 1000 Milli-seconds (*ms*)
- 1 Millisecond = 100 Micro-seconds (μs)
- 1 Microsecond = 1000 Nano-seconds (*ns*)

2.4 Clock Rate

One cycle of a 3GHz cpu takes $\frac{1}{3}$ *ns*

A single execution of `taketime` takes about 25 μs = 75,000 clock cycles.

2.3 The `map` Operation

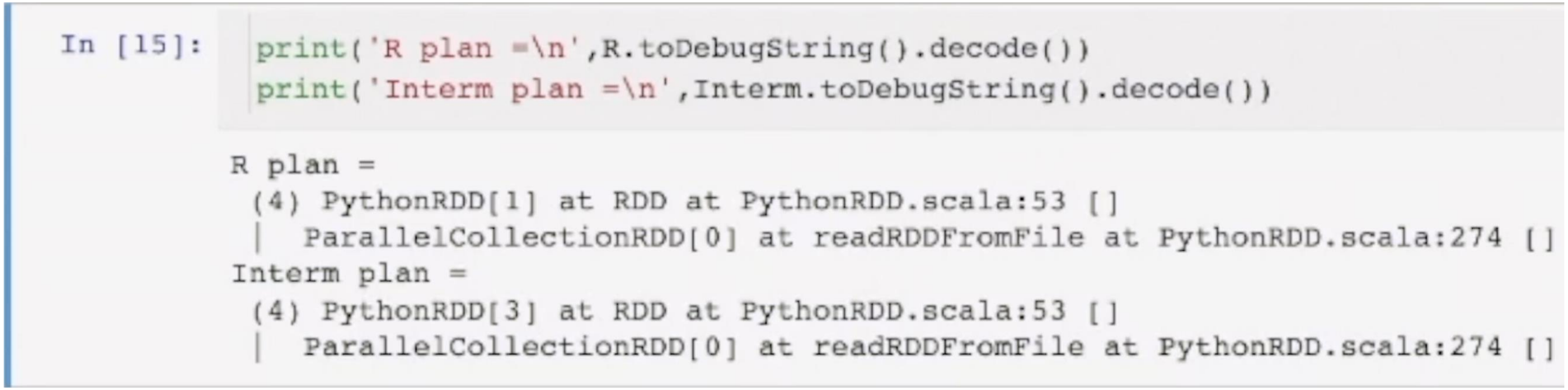


```
In [9]:  %%time
         Interm=R.map(lambda x: taketime(x))

CPU times: user 20 µs, sys: 2 µs, total: 22 µs
Wall time: 24.1 µs
```

2.6 How come so fast?

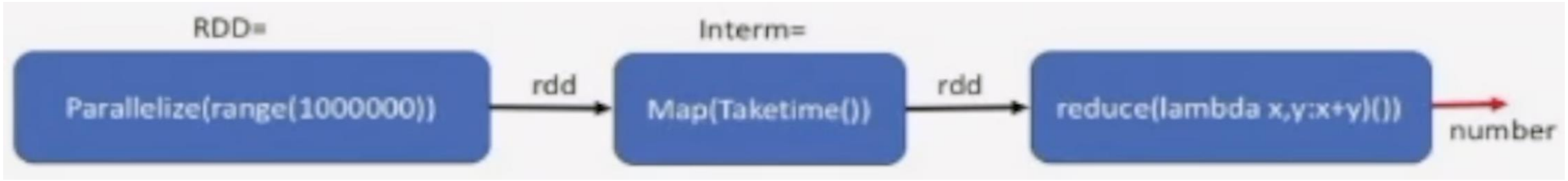
- We expect this map operation to take 1,000,000 * 25 μs = 25 seconds
- **Why** did the previous cell take just 29 μs?

- Because **no computation was done**.
- The cell defined an **execution plan**, but did not execute it yet.



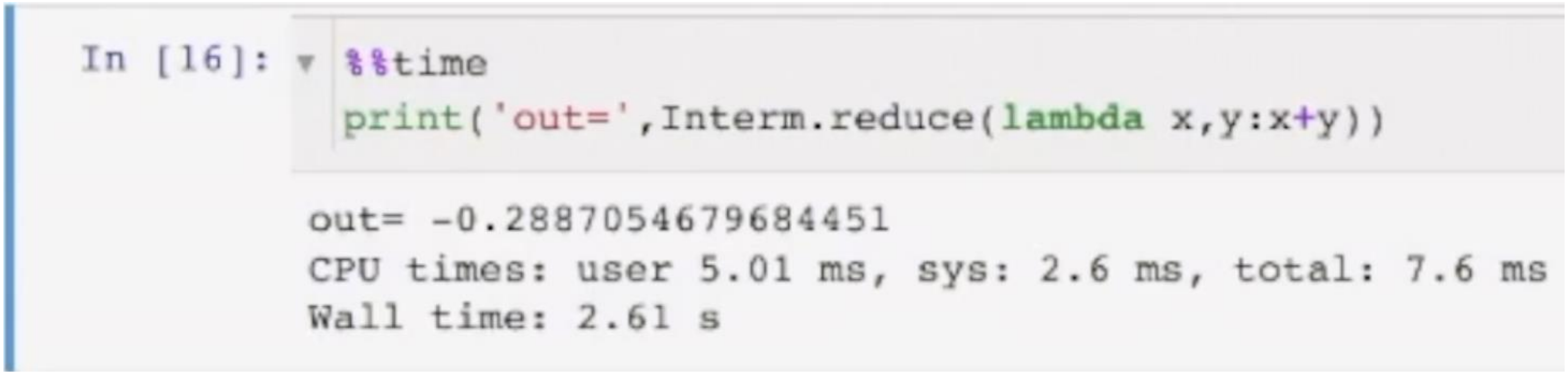
```
In [15]:  print('R plan =\n',R.toDebugString().decode())
          print('Interm plan =\n',Interm.toDebugString().decode())

         R plan =
          (4) PythonRDD[1] at RDD at PythonRDD.scala:53 []
           |  ParallelCollectionRDD[0] at readRDDFromFile at PythonRDD.scala:274 []
         Interm plan =
          (4) PythonRDD[3] at RDD at PythonRDD.scala:53 []
           |  ParallelCollectionRDD[0] at readRDDFromFile at PythonRDD.scala:274 []
```

At this point, only the two left blocks of the plan have been declared.



2.8 Actual Execution

The `reduce` command needs to output an actual output. **Spark** therefore has to actually execute the `map` and the `reduce`. Some real computation needs to be done, which takes about 1-3 seconds (Wall time) depending on the machine used and on its load.



```
In [16]:  %%time
          print('out=',Interm.reduce(lambda x,y:x+y))

          out= -0.2887054679684451
          CPU times: user 5.01 ms, sys: 2.6 ms, total: 7.6 ms
          Wall time: 2.61 s
```

2.9 How come so fast? (Take 2)

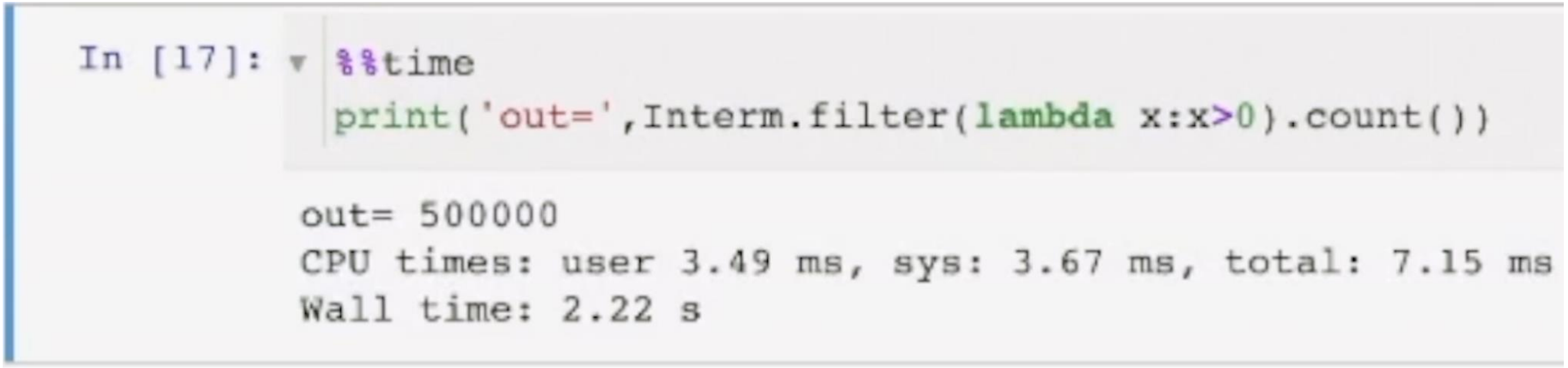
- We expect this map operation to take 1,000,000 * 25 μs = 25 seconds
- Map + reduce takes only ~4 seconds
- Why?

- Because we have **four** workers rather than **one**.
- Because the measurement of a single iteration of `taketime` is an overestimate.

2.10 Executing a different calculation based on the same plan

The plan defined by `Interm` might need to be executed more than once.

Example: Compute the number of map outputs that are larger than zero.

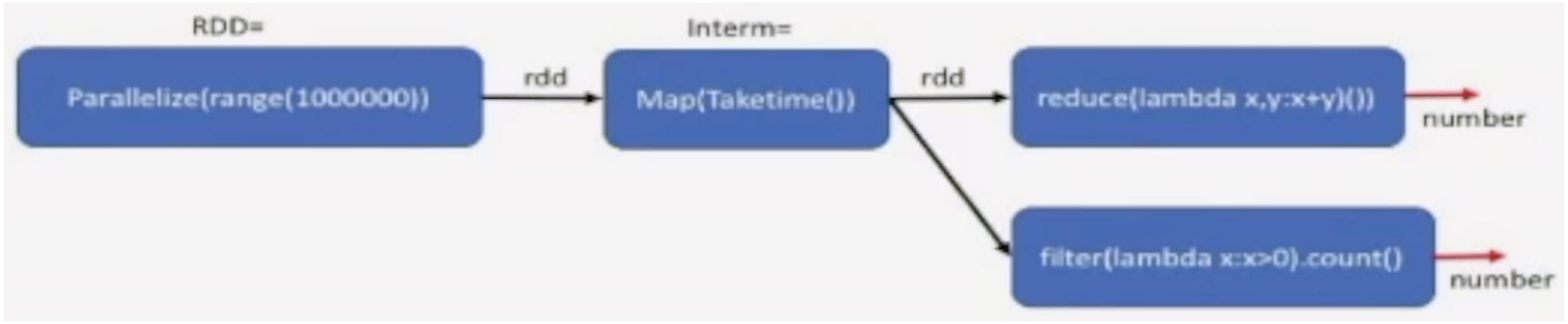


```
In [17]:  %%time
          print('out=',Interm.filter(lambda x:x>0).count())

out= 500000
CPU times: user 3.49 ms, sys: 3.67 ms, total: 7.15 ms
Wall time: 2.22 s
```

2.11 The price of not materializing

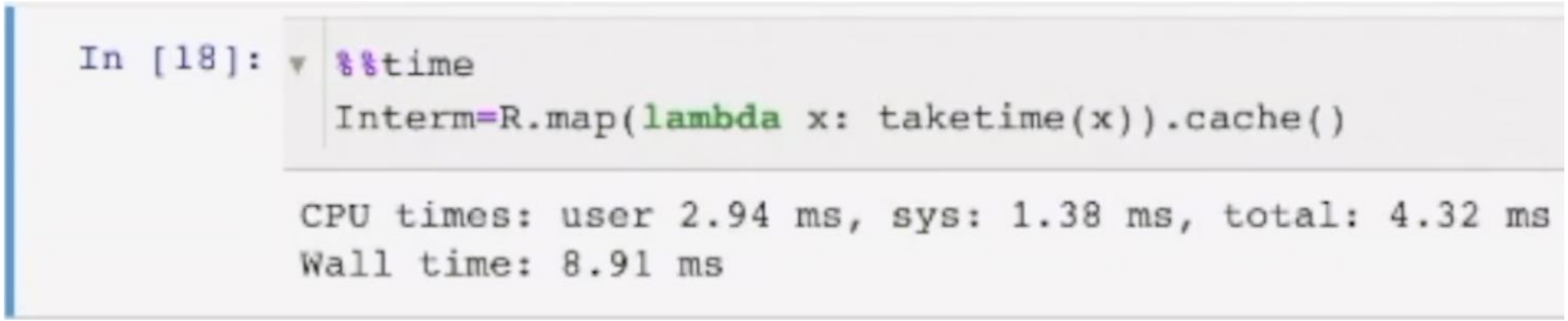
- The run-time (3.4 sec) is similar to that of the reduce (4.4 sec).
- Because the intermediate results in `Interm` have not been saved in memory (materialized).
- They need to be recomputed.

The middle block: `Map(Taketime)` is executed twice, once for each final step.


RDD=
Parallelize(range(1000000))
rdd
Interm=
Map(Taketime())
rdd
reduce(lambda x,y:x+y)())
number
filter(lambda x:x>0).count()
number

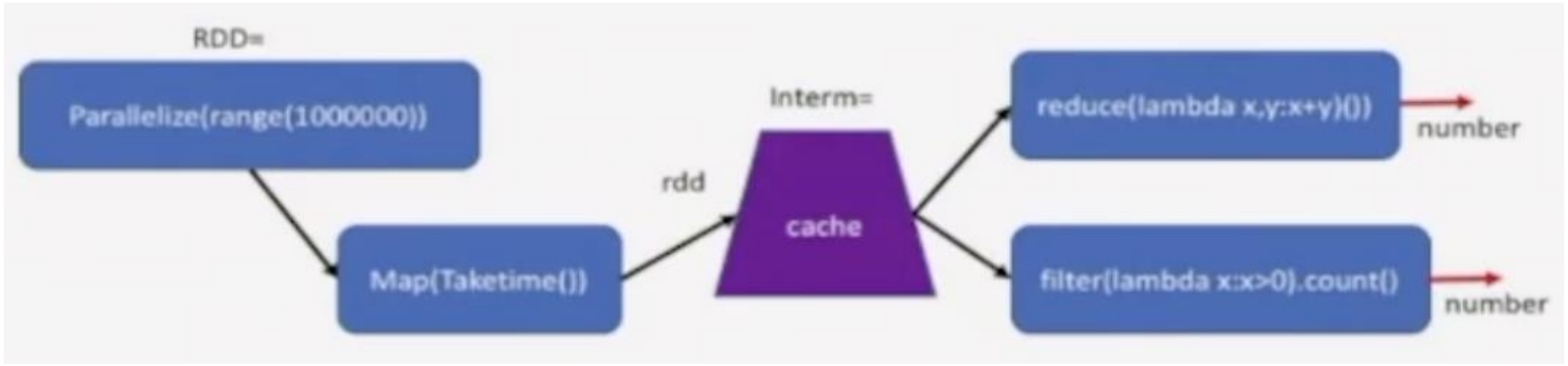
2.12 Caching intermediate results

- We sometimes want to keep the intermediate results in memory so that we can reuse them later without recalculating.
- This will reduce the running time, at the cost of requiring more memory.
- The method `cache()` indicates that the RDD generates in this plan should be stored in memory. Note that this is a **plan to cache**. The actual caching will be done only when the final result is needed.



```
In [18]:  %%time
          Interm=R.map(lambda x: taketime(x)).cache()

          CPU times: user 2.94 ms, sys: 1.38 ms, total: 4.32 ms
          Wall time: 8.91 ms
```

By adding the Cache after `Map(Taketime)`, we save the results of the map for the second computation.

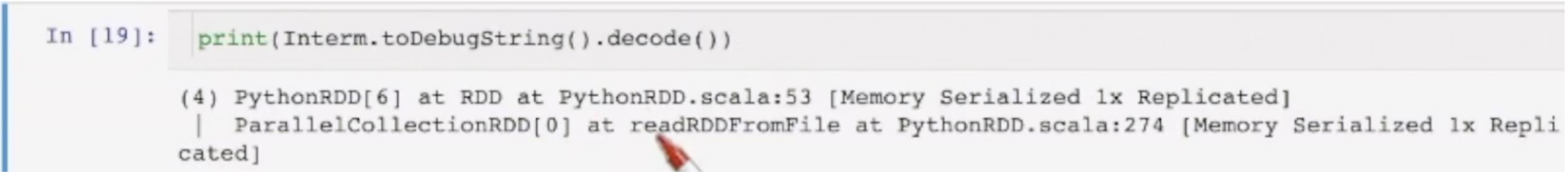

RDD=
Parallelize(range(1000000))
Map(Taketime())
rdd
Interm=
cache
reduce(lambda x,y:x+y)())
number
filter(lambda x:x>0).count()
number

2.13 Plan to cache

The definition of `Interm` is almost the same as before. However, the *plan* corresponding to `Interm` is more elaborate and contains information about how intermediate results will be cached and replicated.

Note that `PythonRDD[4]` is now [Memory Serialized 1x Replicated].

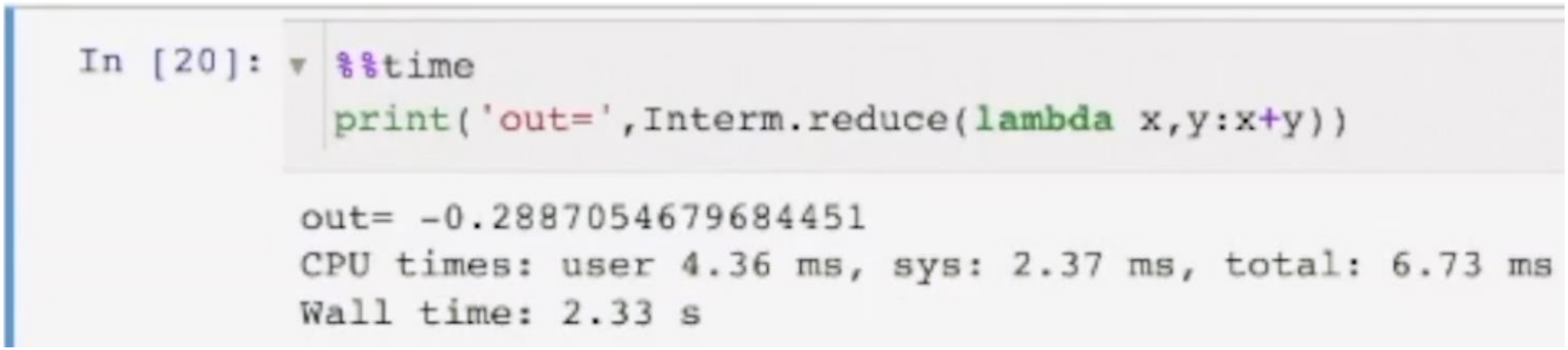
We can check on the plan by applying `.toDebugString()` to the RDD.



```
In [19]:  print(Interm.toDebugString().decode())

          (4) PythonRDD[6] at RDD at PythonRDD.scala:53 [Memory Serialized 1x Replicated]
           |  ParallelCollectionRDD[0] at readRDDFromFile at PythonRDD.scala:274 [Memory Serialized 1x Repli
          cated]
```

2.14 Creating the cache

The following command executes the first map-reduce command **and** caches the result of the `map` command in memory.

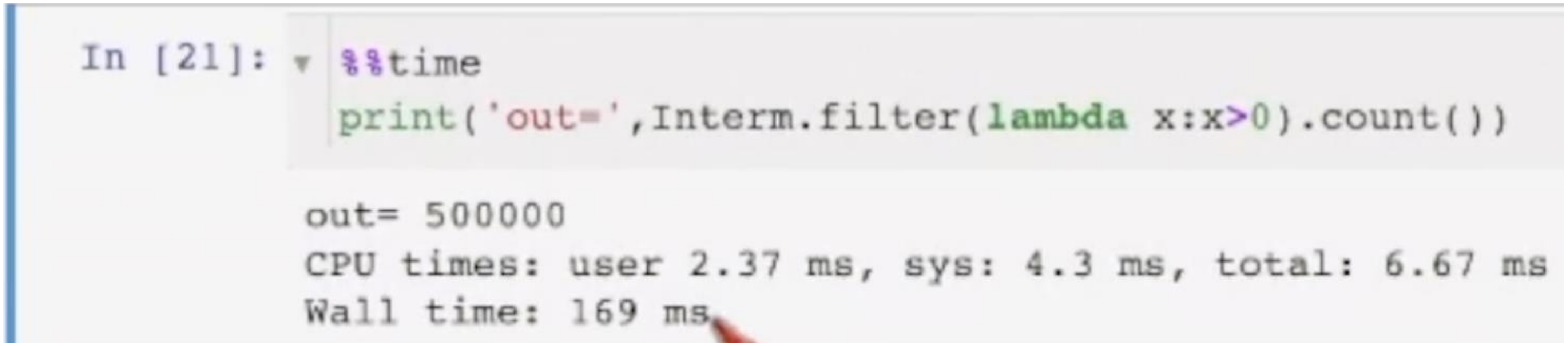


```
In [20]:  %%time
          print('out=',Interm.reduce(lambda x,y:x+y))
```

```
out= -0.2887054679684451
CPU times: user 4.36 ms, sys: 2.37 ms, total: 6.73 ms
Wall time: 2.33 s
```

2.15 Using the cache

This time `Interm` is cached. Therefore, the second use of `Interm` is much faster than when we did not use `cache`: 0.25 second instead of 1.9 second. (Your milage may vary depending on the computer you are running this on).



```
In [21]:  %%time
          print('out=',Interm.filter(lambda x:x>0).count())

          out= 500000
          CPU times: user 2.37 ms, sys: 4.3 ms, total: 6.67 ms
          Wall time: 169 ms
```

3 Summary of evaluation plans

- Spark uses **Lazy Evaluation** to save time and space.
- When the same RDD is needed as input for several computations, it can be better to keep it in memory, also called `cache()`.
- Next Video, Partitioning and Gloming