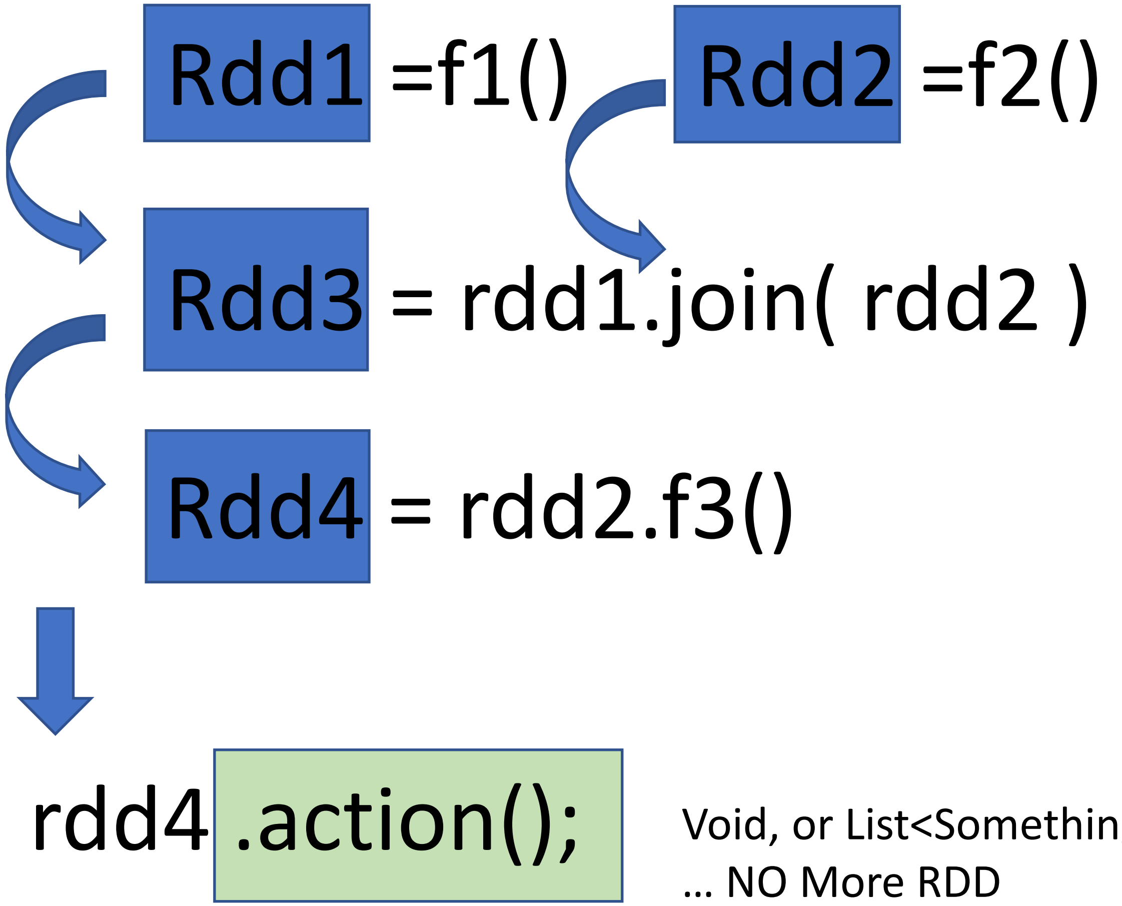
Spark Dataset Lineage Evaluation, Caching & Checkpoint

arnaud.nauwynck@gmail.com

This document:

https://github.com/arnaud-nauwynck/Presentation/big-data/dataset-lineage-eval-caching

Compose Datasets (or RDD) Transformations



Rdd1 =f1()

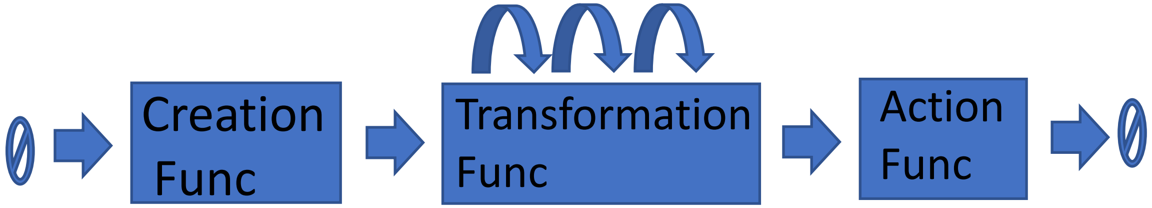
Rdd2 =f2()

Rdd3 = rdd1.join(rdd2)

Rdd4 = rdd2.f3()

rdd4 .action();

Void, or List<SomethingElse>
… NO More RDD



∅

Creation Func

Transformation Func

Action Func

∅

Signature:
(SparkSession, …) -> RDD

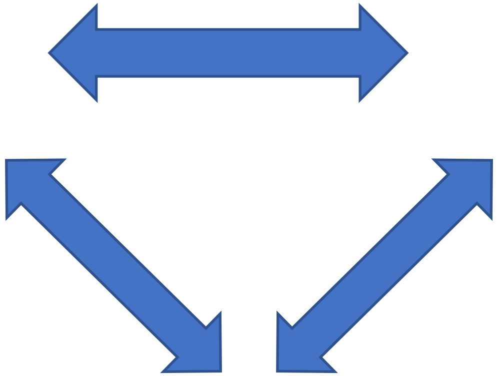
(RDD, …) -> RDD

(RDD, …) -> other

Code Declarations / Expressions / DAG

Variables Declaration
(SSA: Single State Assignement**)**

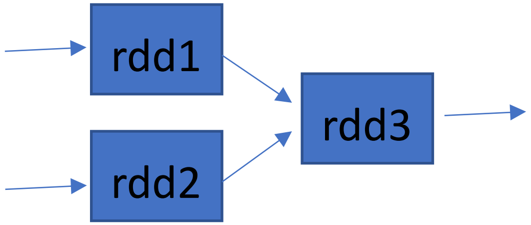
rdd1=..
rdd2=..
rdd3=..



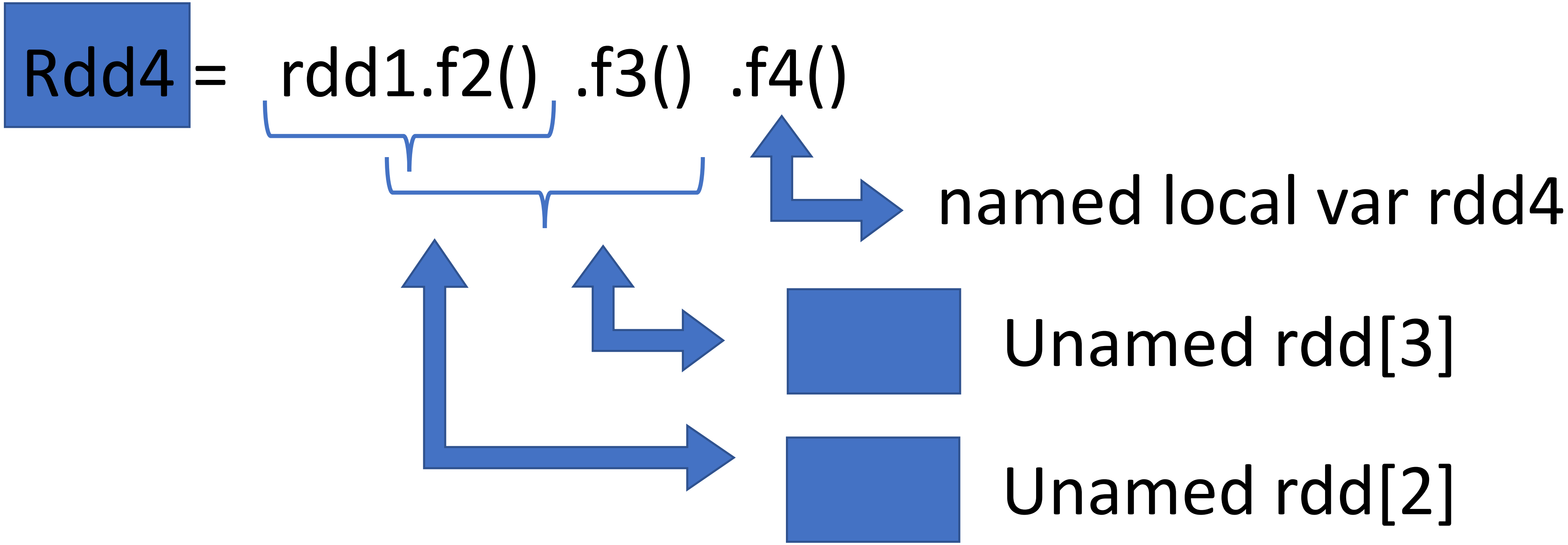
Algebraic Expressions

[f1() .join(f2())].f3()

DAG (Directed Acyclic Graph)



Intermediate (un-named) RDDs using Functionnal API



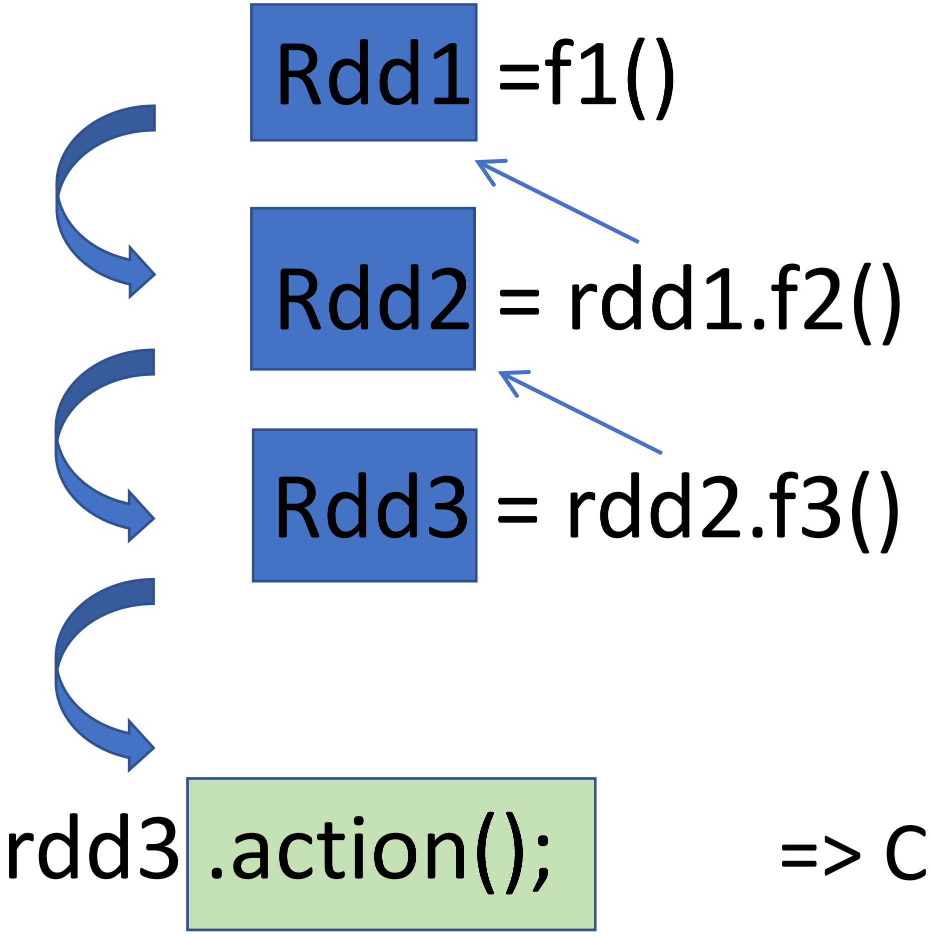
Rdd4 = rdd1.f2() .f3() .f4()

named local var rdd4

Unamed rdd[3]

Unamed rdd[2]

Lazy Transformations … then Action



Rdd1 =f1()

Rdd2 = rdd1.f2()

Rdd3 = rdd2.f3()

rdd3 .action();

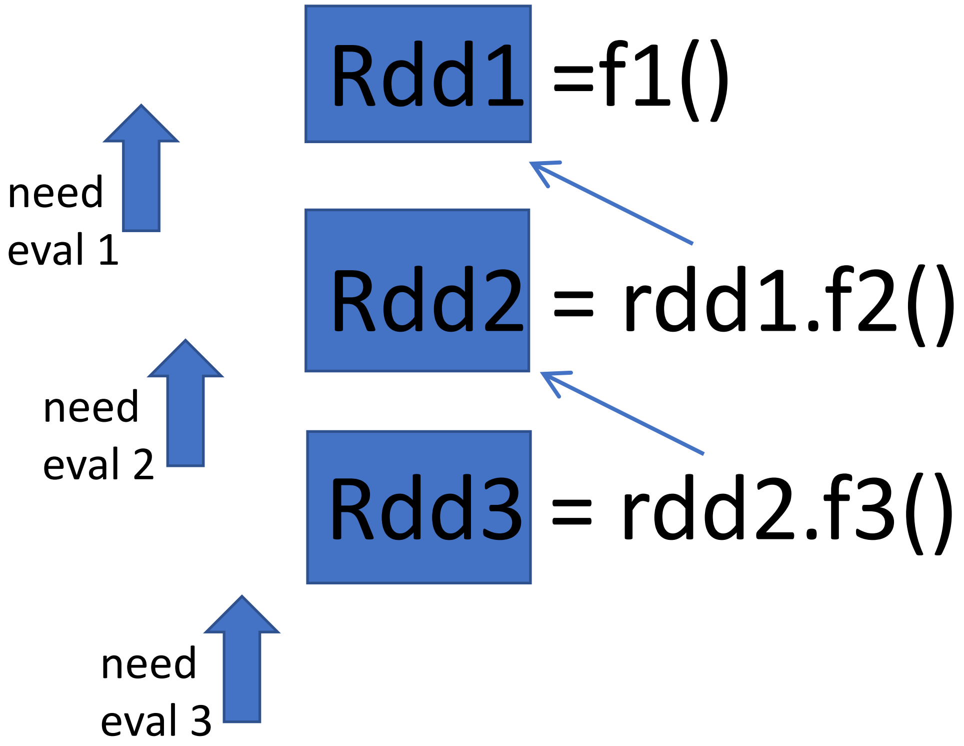
Defined, but lazy
=> nothing computed

=> nothing computed

=> nothing computed

=> Compute all : f3(f2 (f1()))

Compute Backward the Dependency Lineage



Rdd1 =f1()

Rdd2 = rdd1.f2()

Rdd3 = rdd2.f3()

need eval 1

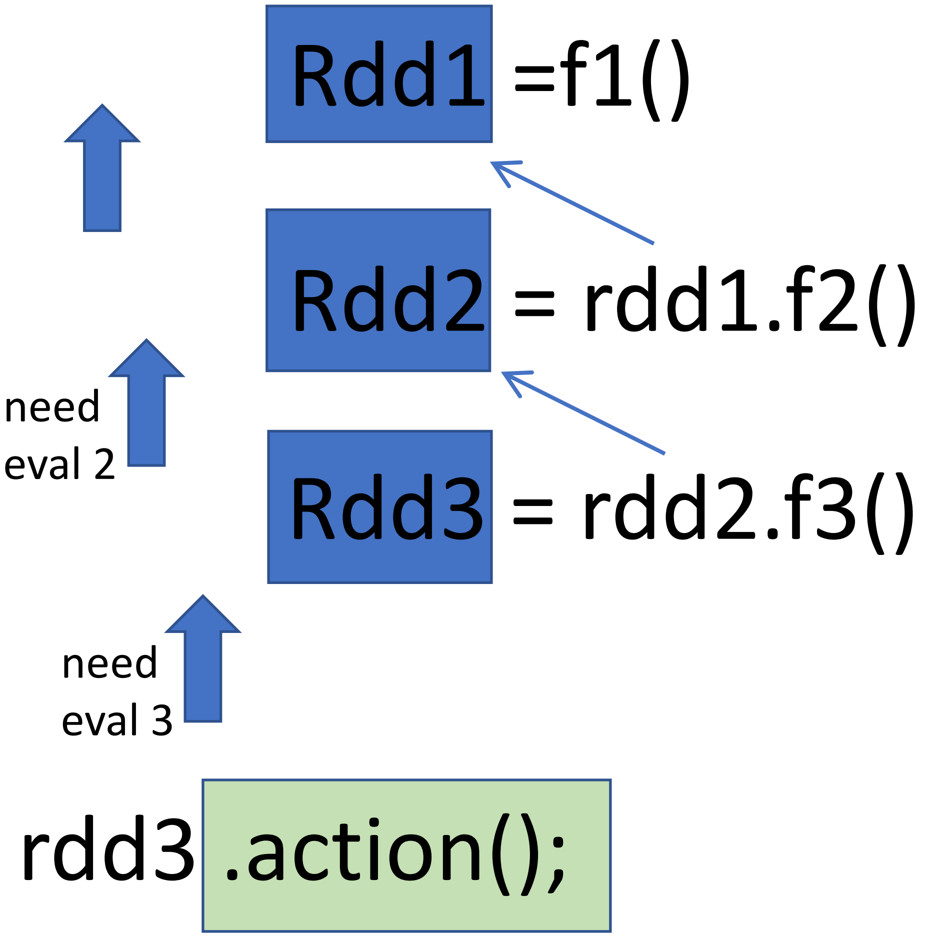
need eval 2

need eval 3

rdd3 .action();

=> Compute all : f3(f2 (f1()))

Every re-executions of RDD
=> re-trigger Full Lineage Recomputation



Rdd1 =f1()

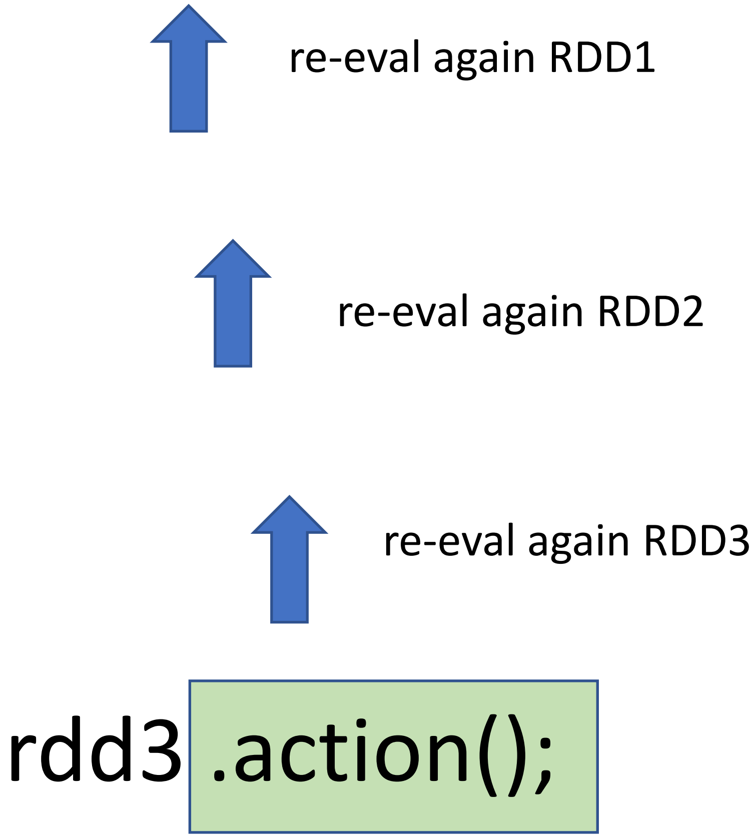
Rdd2 = rdd1.f2()

need
eval 2

Rdd3 = rdd2.f3()

need
eval 3

rdd3 .action();



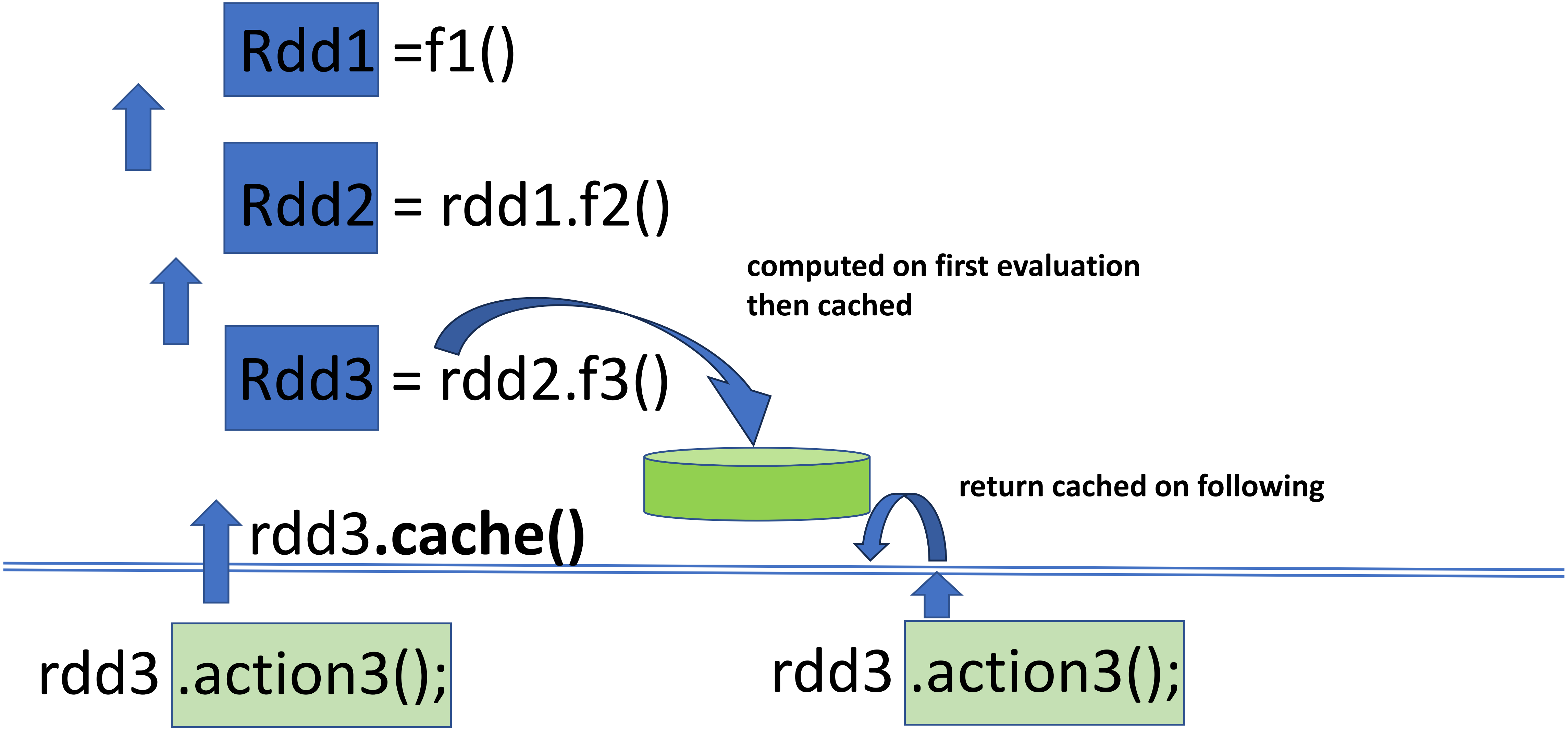
re-eval again RDD1

re-eval again RDD2

re-eval again RDD3

rdd3 .action();

.cache() to Avoid Recomputing



Rdd1 =f1()

Rdd2 = rdd1.f2()

Rdd3 = rdd2.f3()

computed on first evaluation
then cached

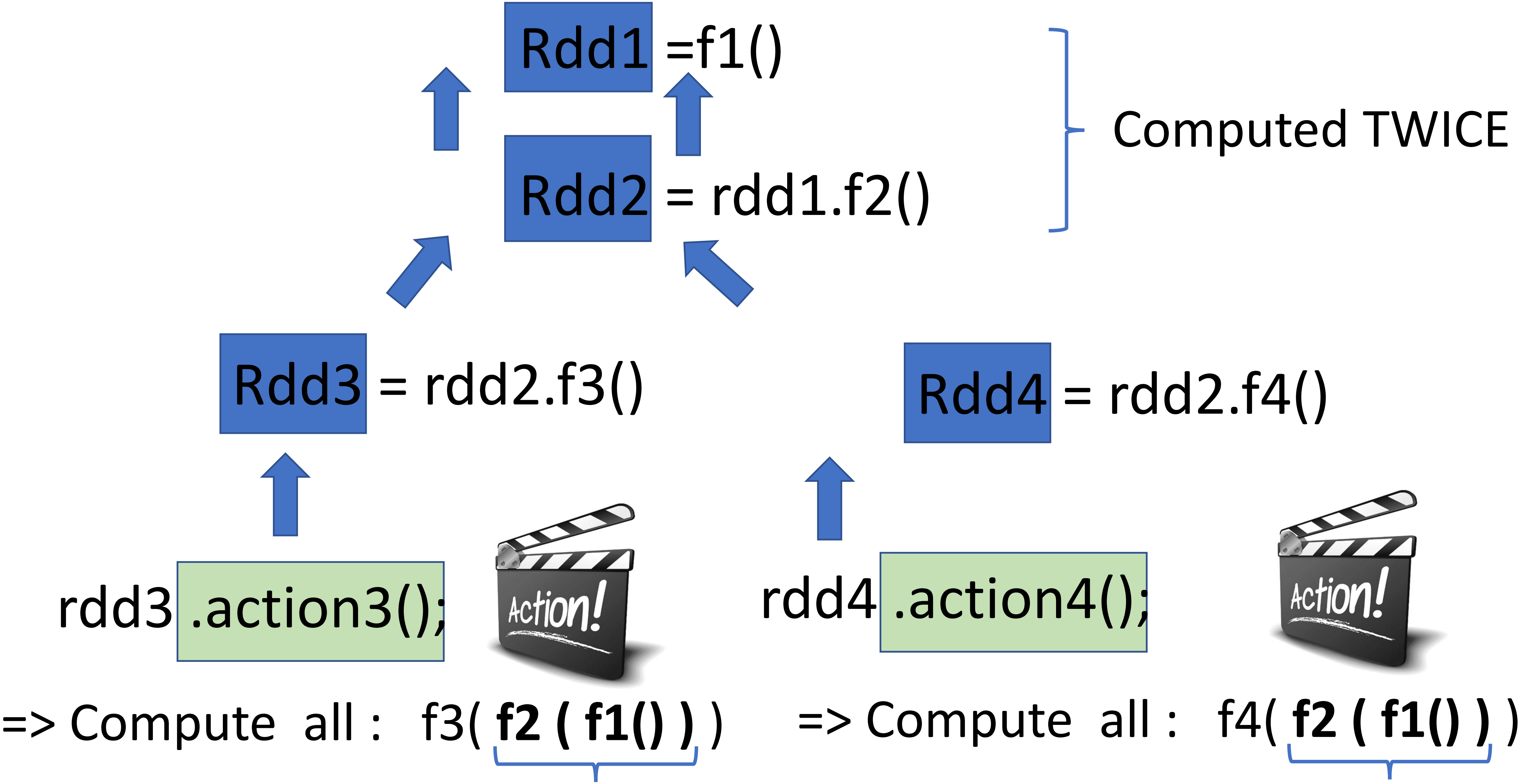
rdd3.**cache()**

return cached on following

rdd3 .action3();

rdd3 .action3();

RDD Shared & Lazy … not necessarily "efficient"



Rdd1 =f1()

Rdd2 = rdd1.f2()

Computed TWICE

Rdd3 = rdd2.f3()

Rdd4 = rdd2.f4()

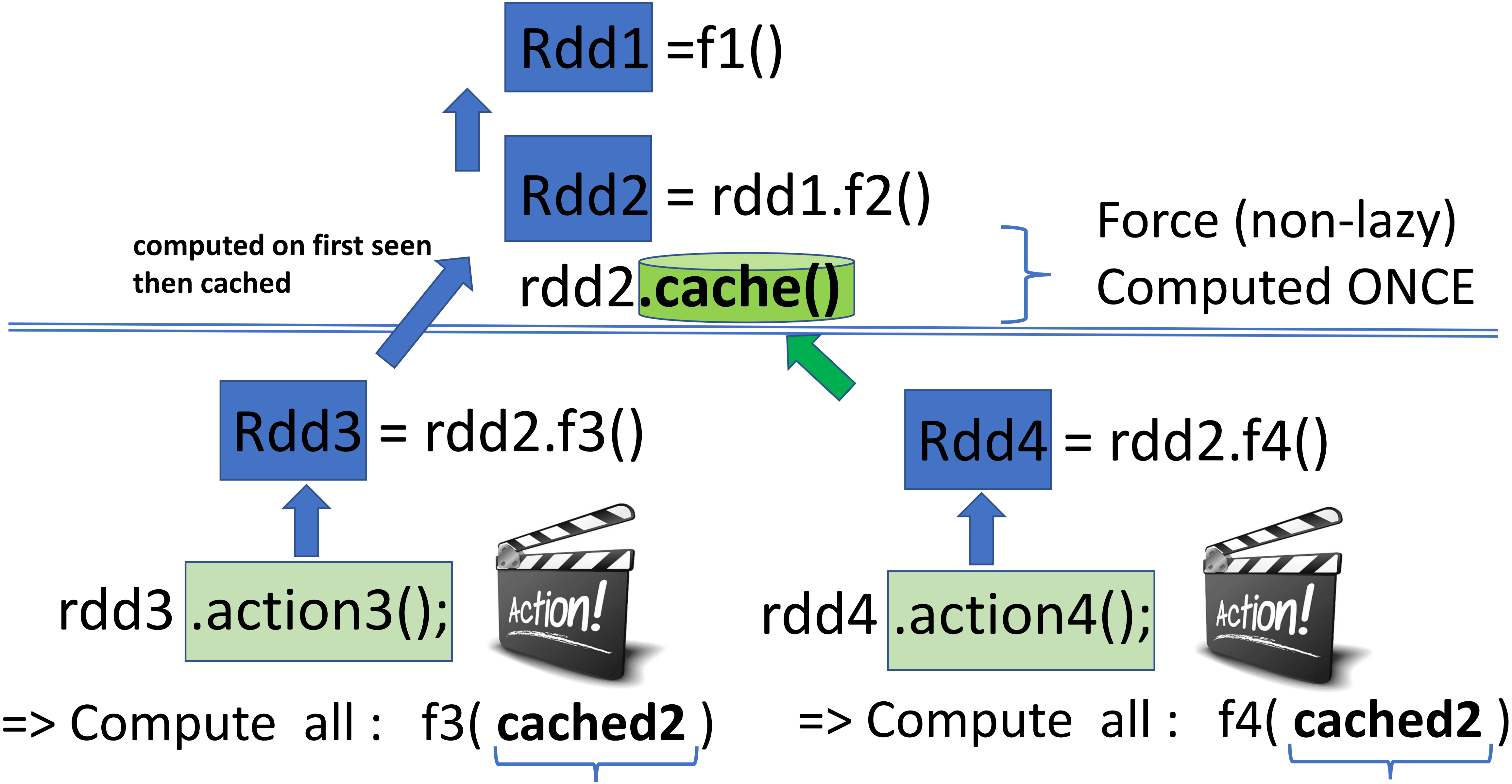
rdd3 .action3();

rdd4 .action4();

=> Compute all : f3(**f2 (f1())**)

=> Compute all : f4(**f2 (f1())**)

Cache common intermediate Result



Rdd1 =f1()

Rdd2 = rdd1.f2()

computed on first seen
then cached

rdd2**.cache()**

Force (non-lazy)
Computed ONCE

Rdd3 = rdd2.f3()

Rdd4 = rdd2.f4()

rdd3 .action3();

rdd4 .action4();

=> Compute all : f3(**cached2**)

=> Compute all : f4(**cached2**)

".cache()" synonym of ".persist()" is special exception in Spark APIs

".cache()" or ".persist() are VERY SPECIAL among all Spark APIs

- RDD/Dataset are immutable
- API return a new transformed RDD/Dataset

BUT ".cache()" is THE exception :

- it modifies a field flag in the RDD
- cache() does not create or return a new object, it reuses the same

Spark Essentials: *Persistence*

transformation	*description*
MEMORY_ONLY	Store RDD as deserialized Java objects in the JVM. If the RDD does not fit in memory, some partitions will not be cached and will be recomputed on the fly each time they're needed. This is the default level.
MEMORY_AND_DISK	Store RDD as deserialized Java objects in the JVM. If the RDD does not fit in memory, store the partitions that don't fit on disk, and read them from there when they're needed.
MEMORY_ONLY_SER	Store RDD as serialized Java objects (one byte array per partition). This is generally more space-efficient than deserialized objects, especially when using a fast serializer, but more CPU-intensive to read.
MEMORY_AND_DISK_SER	Similar to MEMORY_ONLY_SER, but spill partitions that don't fit in memory to disk instead of recomputing them on the fly each time they're needed.
DISK_ONLY	Store the RDD partitions only on disk.
MEMORY_ONLY_2, **MEMORY_AND_DISK_2,** etc	Same as the levels above, but replicate each partition on two cluster nodes.

Memory (and Disk) are very Limited Executor Resources

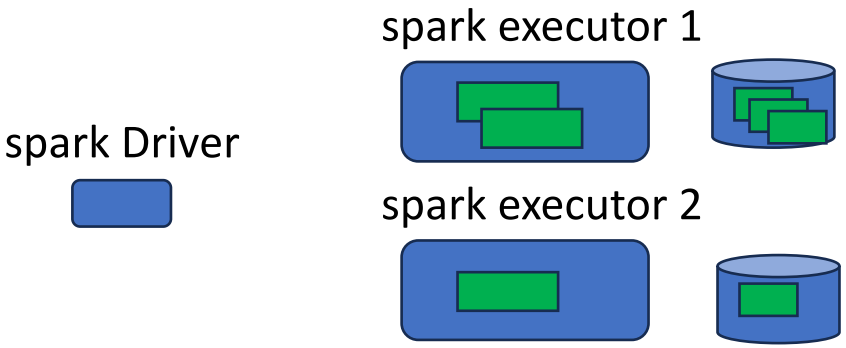
you can not cache "too much" data.
Do only when re-using may be faster than recomputing.

when not enough MEMORY => will use "DISK"

Disk may be slower than recomputing / re-fetching the source itself

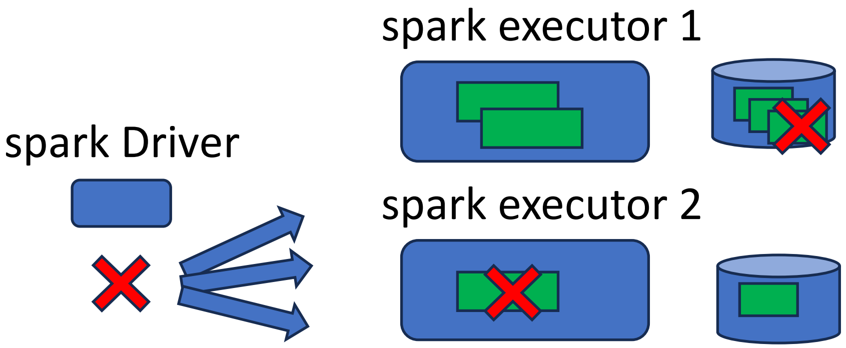
Memory and Disk are per executors (per block files)

```
{
  val ds = ... ;
  **ds.cache()**
  .. usage1(ds)
  .. usage2(ds)
}
```



Do you need to ".unpersist()" ?
What if you don't ?

```
{
   val ds = ... ;
   ds.cache()
   .. usage1(ds)
   .. usage2(ds)
}
```



thread variable "ds" is NOT more accessible after "}"

The JVM GarbageCollector can reclaim "ds" memory on driver

=> spark will automatically release corresponding blocks (MEMORY / DISK) on all executors

PROBLEM: this is GC on driver-side. The driver may have plenty of free memory, so GC might not be triggered soon !
... BUT executor may be short of memory

When choosing explicitly ".unpersist()", when to do it ?

```
val ds = ....
ds.persist();
try {

    ds.action1();
    ds.action2();

} finally {
   ds.unpersist();
}

ds.action3();
```

OK
ds is cached inside
try-finally code

Problem !!
ds is not cached anymore
action will re-compute FULL Lineage

Difficulty Tracking "hidden" Dependency References in Code

Looks OK
ds is cached inside
try-finally code

```
Dataset transformedDs;

{
  val ds = ....
  ds.persist();
 try {

    ds.action1();
    transformedDs = ds.map(..);
    transformedDs.action();

 } finally {
   ds.unpersist();
 }
} // OK, ds not visible outside of method code
 // code would not compile
return transformedDs; // use later
```

Problem !!
transformedDs refers to ds,
which is not cached anymore
action will re-compute FULL Lineage

Caching - Display Result in Spark UI

when using
dataset.cache()

The **FULL Dag Lineage** is still displayed in Spark UI > Query

Some Job have 0 Stage, marked as "Skipped"

Caching - Display Result in Spark UI

Using .localCheckpoint() instead of cache()

when using

Dataset replacedDs = dataset.localCheckpoint()

equivalent to use cached block in-memory & files

The **Dag Lineage is CUT** in Spark UI > Query.
the source is displayed as "InMemoryRDD"

=> much easier to read (even if not named/labelled correctly)

Using .checkpoint()

when using

Dataset replacedDs = dataset.checkpoint()

equivalent to use saved files in persistent Hadoop storage

The **Dag Lineage is CUT** in Spark UI > Query.
the source is displayed as "FileScanRDD"

=> much easier to read (even if not named/labelled correctly)

Questions ?