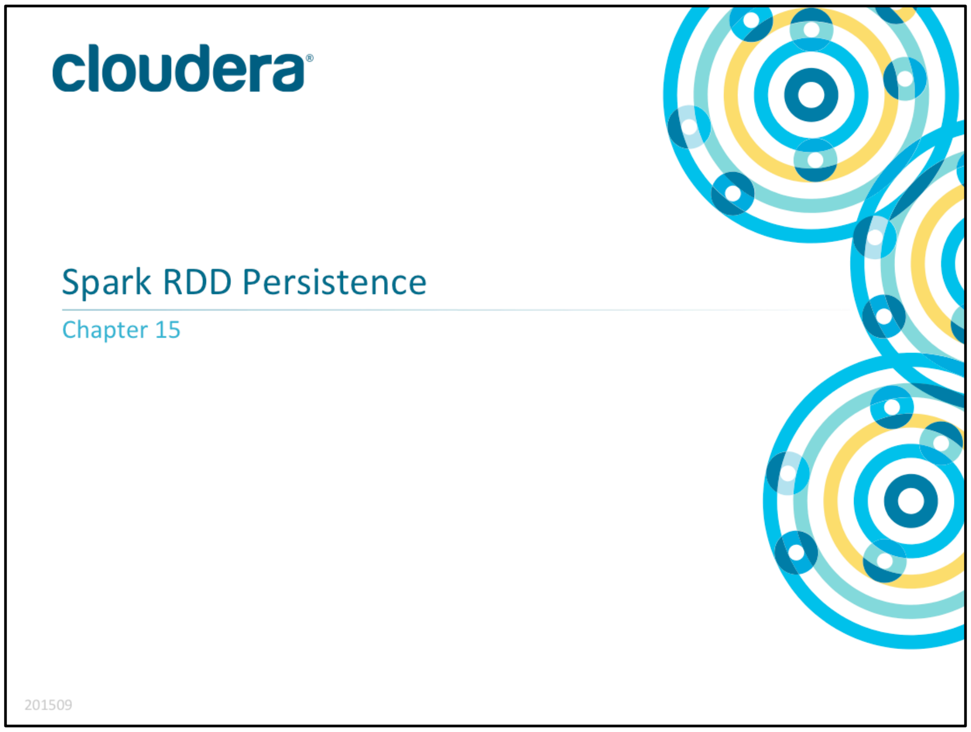


Spark RDD Persistence

Chapter 15

201509

Course Chapters

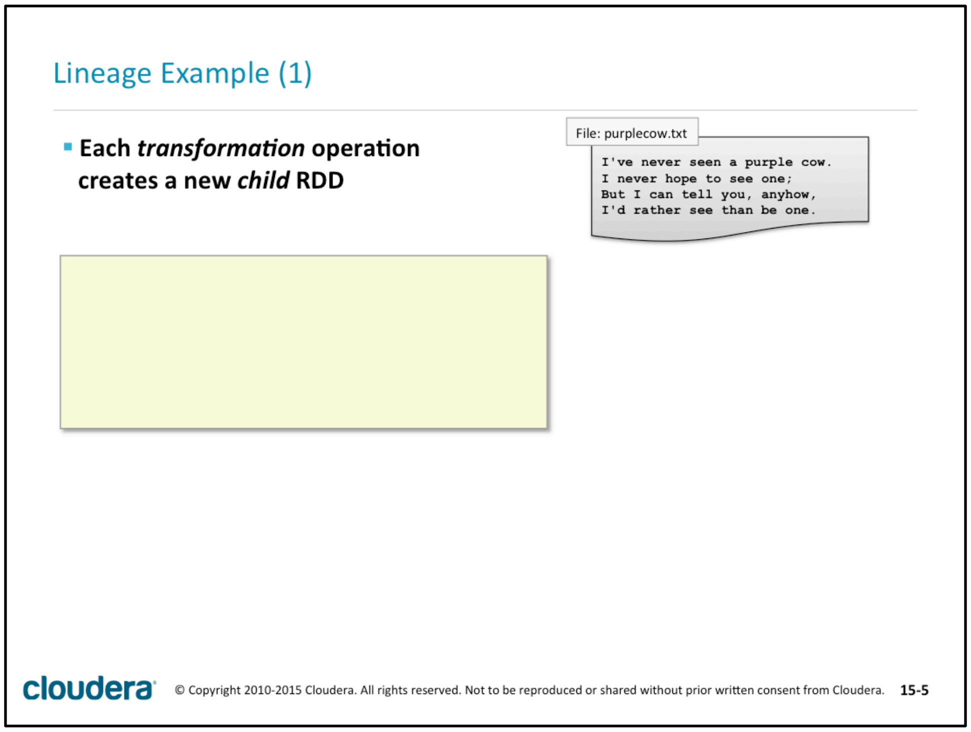
Spark RDD Persistence

In this chapter you will learn

- **How Spark uses an RDD's lineage in operations**
- **How to persist RDDs to improve performance**

Chapter Topics

Spark RDD Persistence | **Distributed Data Processing with Spark**



Lineage Example (1)

- **Each *transformation* operation creates a new *child* RDD**

File: purplecow.txt

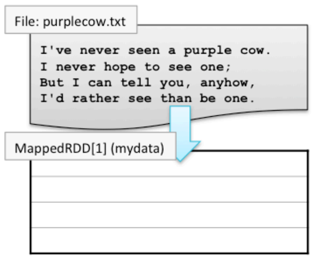
```
I've never seen a purple cow.
I never hope to see one;
But I can tell you, anyhow,
I'd rather see than be one.
```

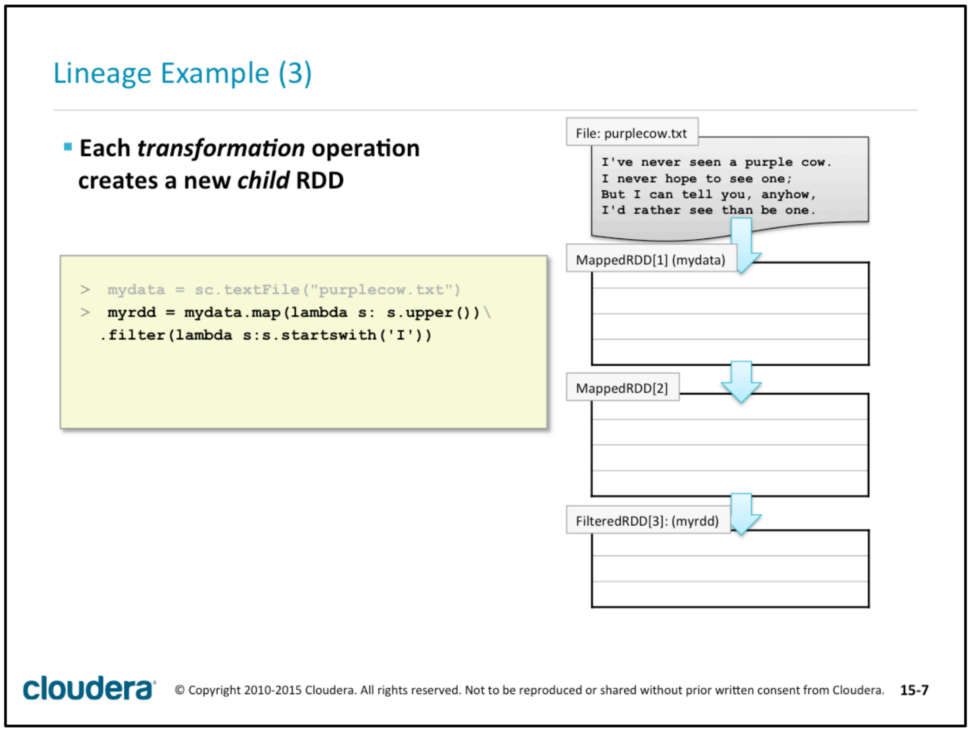
START BUILD
This code example is the same one we used earlier in class: read a file, convert to upper case, filter, and count. We are re-using here to demonstrate the concept of lineage.

Lineage Example (2)

- Each *transformation* operation creates a new *child* RDD

```
> mydata = sc.textFile("purplecow.txt")
```

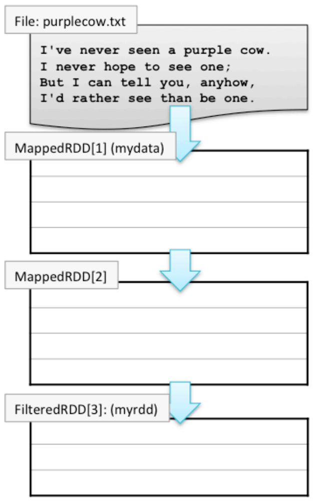

File: purplecow.txt
I've never seen a purple cow.
I never hope to see one;
But I can tell you, anyhow,
I'd rather see than be one.
MappedRDD[1] (mydata)



Lineage Example (3)

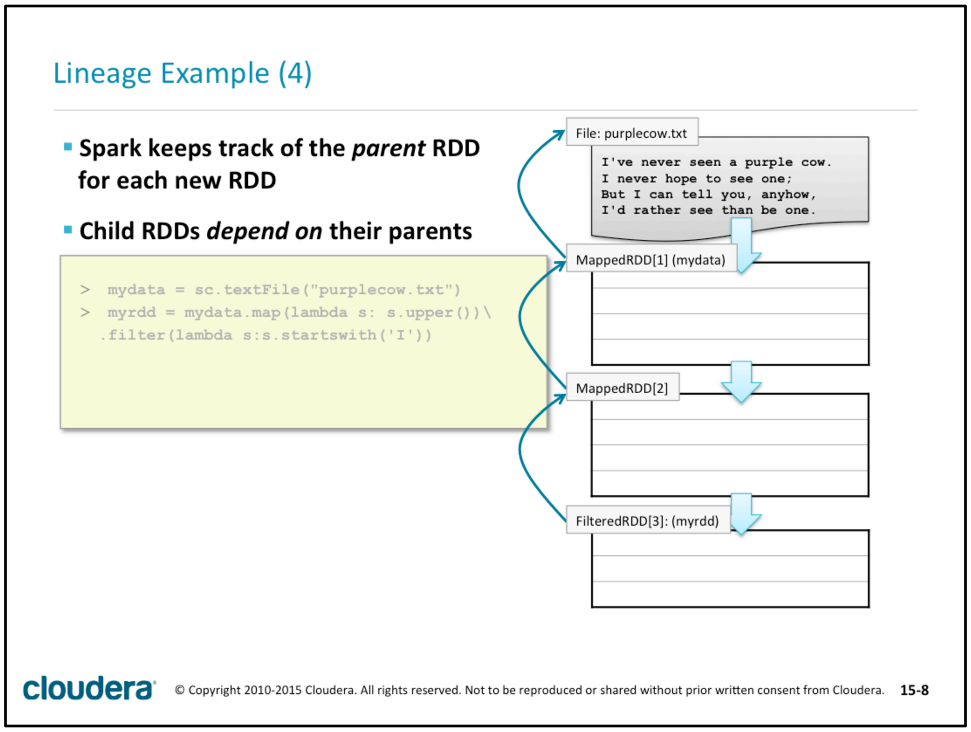
- **Each *transformation* operation creates a new *child* RDD**

```
> mydata = sc.textFile("purplecow.txt")
> myrdd = mydata.map(lambda s: s.upper())\
  .filter(lambda s:s.startswith('I'))
```



Important point, many students get confused regarding the difference between the name of an RDD (e.g. FilteredRDD[3]), and the name of the variable we are using to refer to the RDD (e.g. "myrdd").

In this example we are showing Spark's RDD name (MappedRDD, etc.) and in parentheses the variable the program uses to refer to it. Note that the second one has no variable, because it's an intermediate value as a chain…but it *still has a Spark identifier*.



Lineage Example (4)

- Spark keeps track of the *parent* RDD for each new RDD
- Child RDDs *depend on* their parents

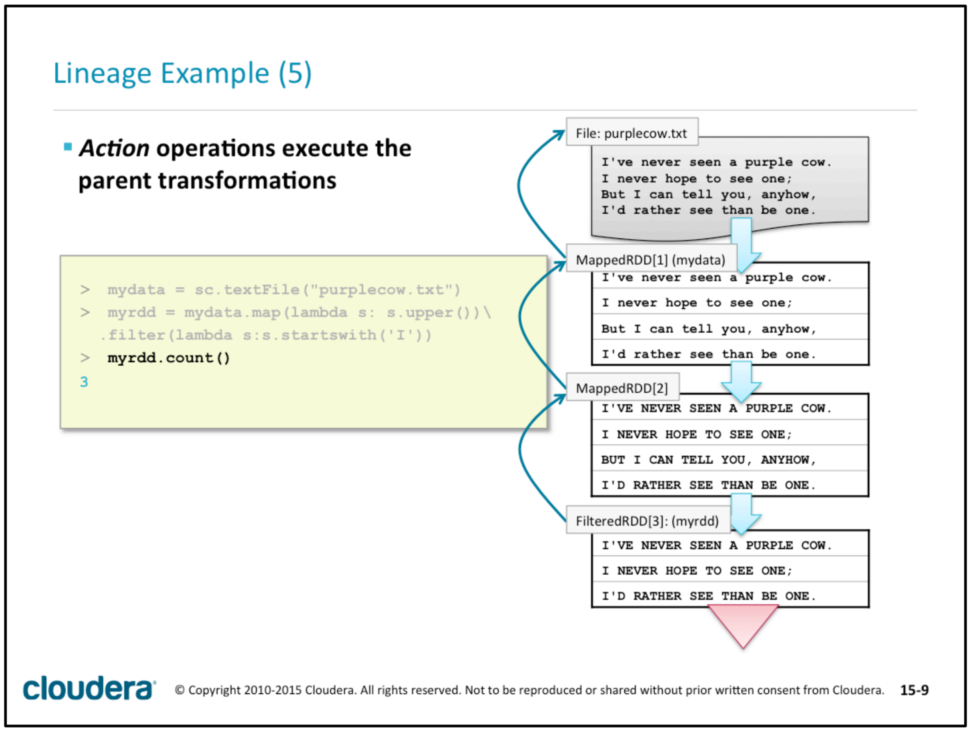
```
> mydata = sc.textFile("purplecow.txt")
> myrdd = mydata.map(lambda s: s.upper())\
  .filter(lambda s:s.startswith('I'))
```

The blue arrows show the dependency of the lineage. RDD 3 depends on RDD 2, which depends on RDD 1, which depends on the source file.

This lineage is stored as part of each RDD.

You can see an RDD's lineage using the toDebugString function.

Instructor side-note on toDebugString:
There's a difference in how Python and Scala store lineages. In this example, Scala would show a lineage as implied by this slide, of 4 RDDs. Python, however, doesn't materialize intermediate RDDs if it doesn't need to – that is, if they can be pipelined together, as all the examples in this slide can be, because they are "narrow" (map) operations. So Python results may be confusing when compared side by side with Scala. This is particularly true later when we get to iterative RDDs and checkpoints.



Lineage Example (5)

- ***Action*** **operations execute the parent transformations**

```
> mydata = sc.textFile("purplecow.txt")
> myrdd = mydata.map(lambda s: s.upper())\
  .filter(lambda s:s.startswith('I'))
> myrdd.count()
3
```

File: purplecow.txt

```
I've never seen a purple cow.
I never hope to see one;
But I can tell you, anyhow,
I'd rather see than be one.
```

MappedRDD[1] (mydata)

```
I've never seen a purple cow.
I never hope to see one;
But I can tell you, anyhow,
I'd rather see than be one.
```

MappedRDD[2]

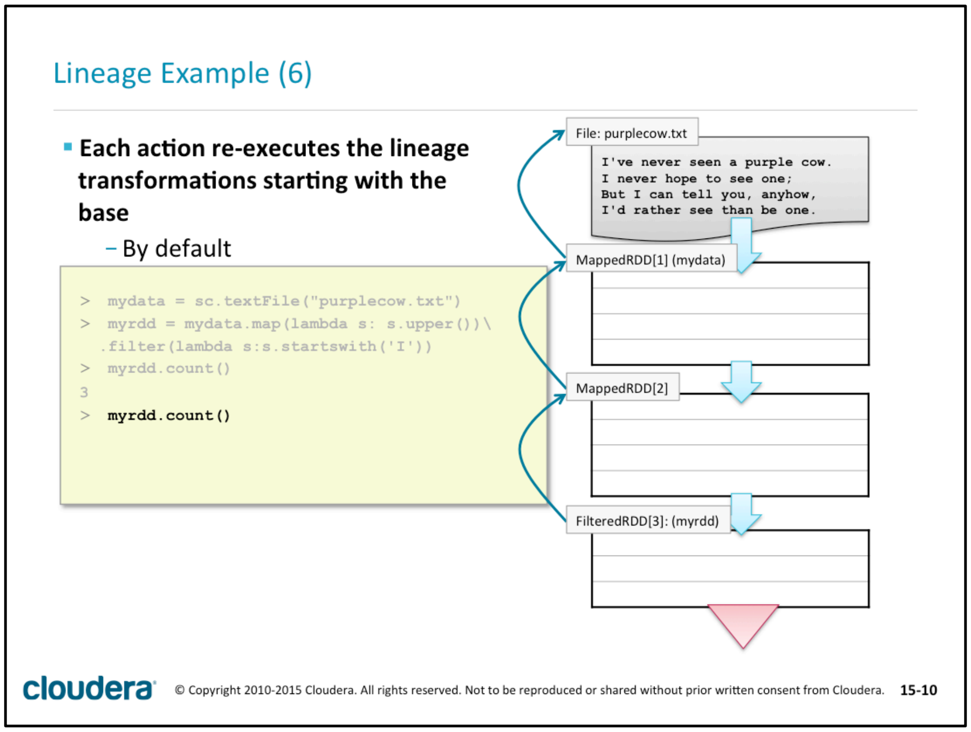
```
I'VE NEVER SEEN A PURPLE COW.
I NEVER HOPE TO SEE ONE;
BUT I CAN TELL YOU, ANYHOW,
I'D RATHER SEE THAN BE ONE.
```

FilteredRDD[3]: (myrdd)

```
I'VE NEVER SEEN A PURPLE COW.
I NEVER HOPE TO SEE ONE;
I'D RATHER SEE THAN BE ONE.
```

cloudera

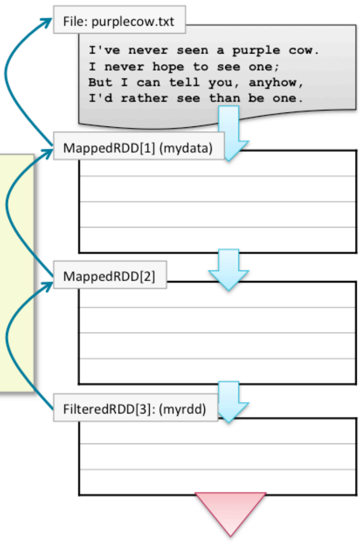
When count is called, it triggers computation of FilteredRDD3, which in turn triggers computation of its parent, MappedRDD2, and so on. So at that point, each RDD recursively computes its parents.



Lineage Example (6)

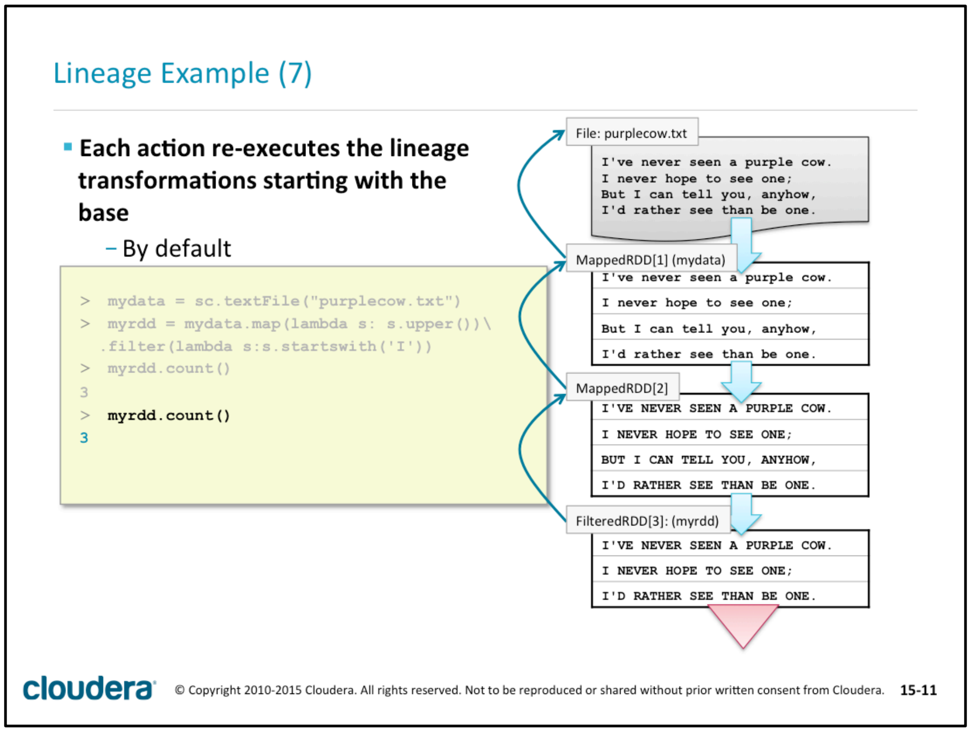
- **Each action re-executes the lineage transformations starting with the base**
  - By default

```
> mydata = sc.textFile("purplecow.txt")
> myrdd = mydata.map(lambda s: s.upper())\
 .filter(lambda s:s.startswith('I'))
> myrdd.count()
3
> myrdd.count()
```



So, in the last slide, we called count() and got output = 3.

What happens if we call it again?



Lineage Example (7)

- **Each action re-executes the lineage transformations starting with the base**
  - By default

```
> mydata = sc.textFile("purplecow.txt")
> myrdd = mydata.map(lambda s: s.upper())\
 .filter(lambda s:s.startswith('I'))
> myrdd.count()
3
> myrdd.count()
3
```

File: purplecow.txt

```
I've never seen a purple cow.
I never hope to see one;
But I can tell you, anyhow,
I'd rather see than be one.
```

MappedRDD[1] (mydata)

```
I've never seen a purple cow.
I never hope to see one;
But I can tell you, anyhow,
I'd rather see than be one.
```

MappedRDD[2]

```
I'VE NEVER SEEN A PURPLE COW.
I NEVER HOPE TO SEE ONE;
BUT I CAN TELL YOU, ANYHOW,
I'D RATHER SEE THAN BE ONE.
```

FilteredRDD[3]: (myrdd)

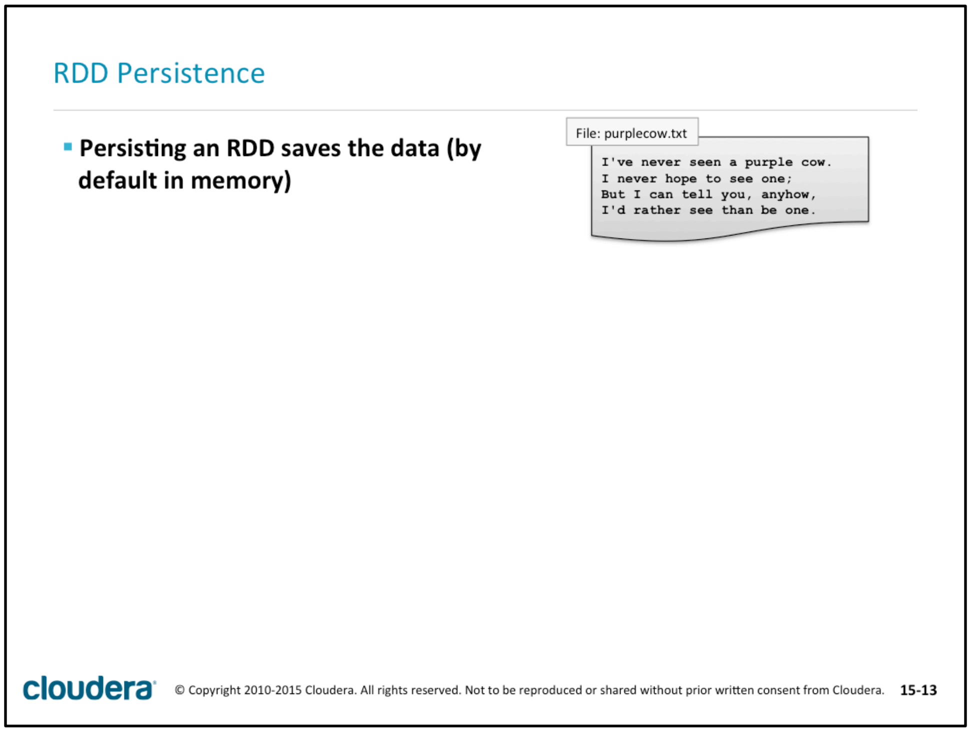
```
I'VE NEVER SEEN A PURPLE COW.
I NEVER HOPE TO SEE ONE;
I'D RATHER SEE THAN BE ONE.
```

END BUILD
By default, the second time you call count() (or any action) on myrdd, it will traverse the RDD lineage tree and re-execute the transformations.

Chapter Topics

Spark RDD Persistence	Distributed Data Processing with Spark

- RDD Lineage
- **RDD Persistence Overview**
- Distributed Persistence
- Conclusion
- Homework: Persist an RDD



RDD Persistence

- **Persisting an RDD saves the data (by default in memory)**

File: purplecow.txt

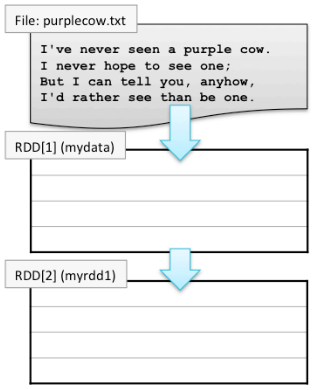
```
I've never seen a purple cow.
I never hope to see one;
But I can tell you, anyhow,
I'd rather see than be one.
```

START BUILD
same example as before, but this time with persistence!

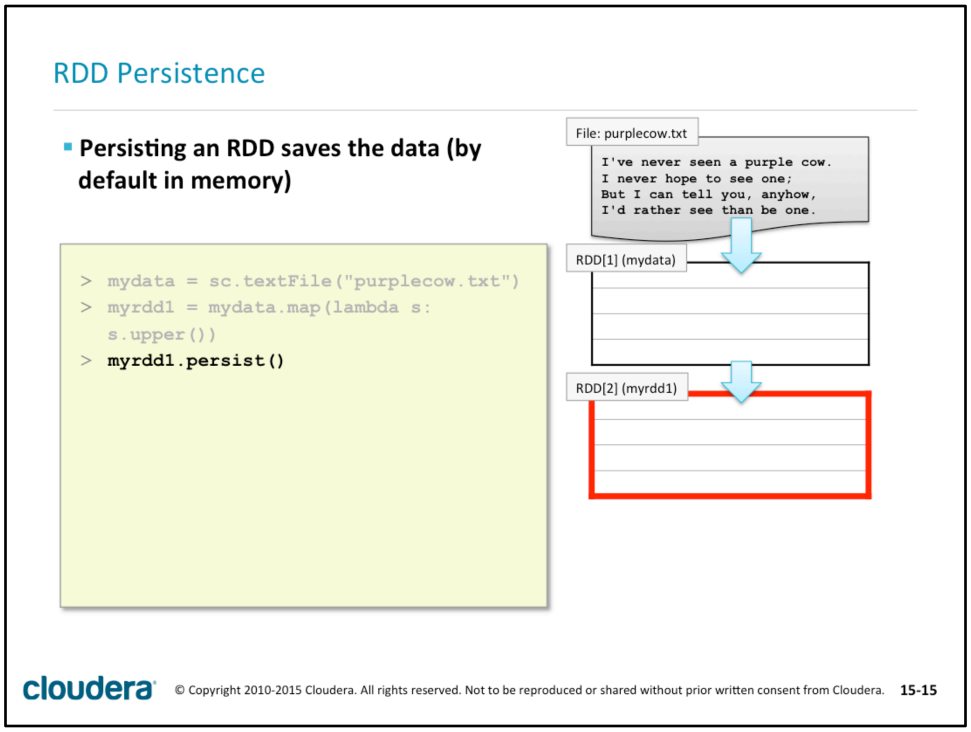
RDD Persistence

- **Persisting an RDD saves the data (by default in memory)**

```
> mydata = sc.textFile("purplecow.txt")
> myrdd1 = mydata.map(lambda s:
  s.upper())
```


File: purplecow.txt
I've never seen a purple cow.
I never hope to see one;
But I can tell you, anyhow,
I'd rather see than be one.
RDD[1] (mydata)
RDD[2] (myrdd1)

This code example is in Persistence.pyspark. The Scala example is in Persistence.scalaspark



RDD Persistence

- **Persisting an RDD saves the data (by default in memory)**

```
> mydata = sc.textFile("purplecow.txt")
> myrdd1 = mydata.map(lambda s:
  s.upper())
> myrdd1.persist()
```

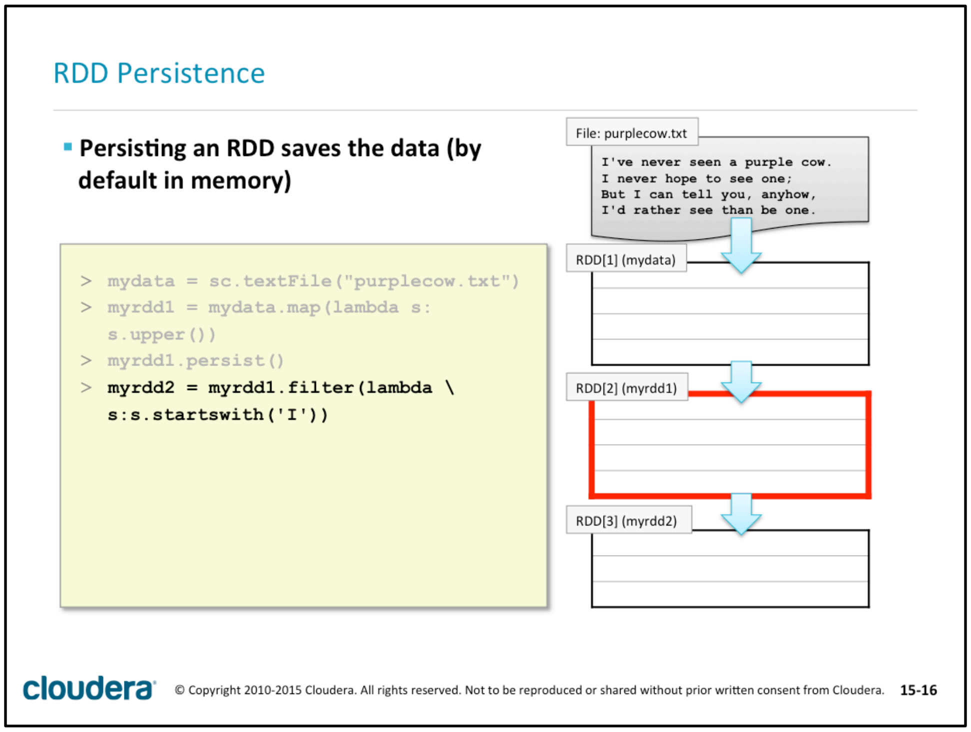
File: purplecow.txt
I've never seen a purple cow.
I never hope to see one;
But I can tell you, anyhow,
I'd rather see than be one.
RDD[1] (mydata)
RDD[2] (myrdd1)

The red border on RDD 2 (myrdd) indicates a persisted RDD.

Note that calling persist or cache does not trigger execution/computation. Instead, it marks the RDD data to be stored when next it is computed. But for now, the diagram shows it empty, because it hasn't been computed this time around.

Instructor side note: note that unlike in the last example, we've now assigned a variable name to the intermediate RDD (RDD2), to make the example easier to read. This can be confusing because the name of the RDDs *do not match* the variable names. the variable myrdd2 points to RDD3. This was a deliberate choice, to help students understand the difference between the variable name, and the RDD id, a common source of confusion.

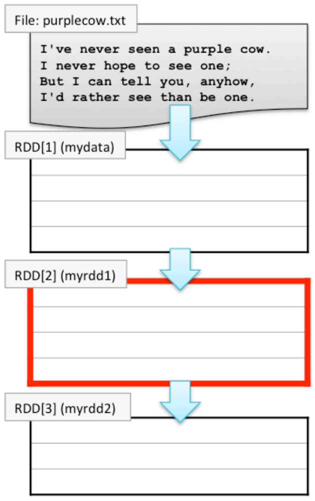
In previous versions of these slides .cache() was used instead of .persist. .cache is really just a shortcut for .persist(MEMORY_ONLY), we introduce .cache() later and simplify the concept by avoiding the word cache. We can persist in memory only, persist in memory and spilling to disk, and persist onto disk only.



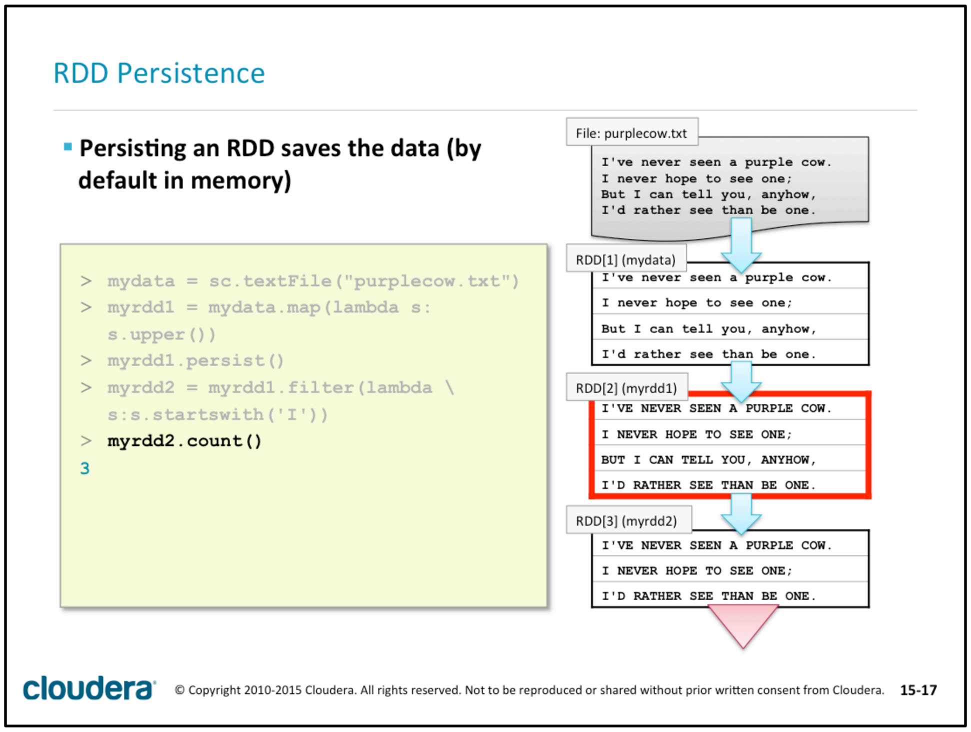
RDD Persistence

- **Persisting an RDD saves the data (by default in memory)**

```
> mydata = sc.textFile("purplecow.txt")
> myrdd1 = mydata.map(lambda s:
  s.upper())
> myrdd1.persist()
> myrdd2 = myrdd1.filter(lambda \
  s:s.startswith('I'))
```



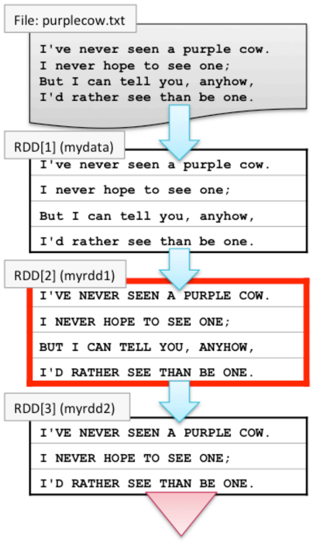
We still haven't computed anything, we've just set up the same lineage we did before, except that the second RDD in the lineage is marked to be persisted/cached (red border).



RDD Persistence

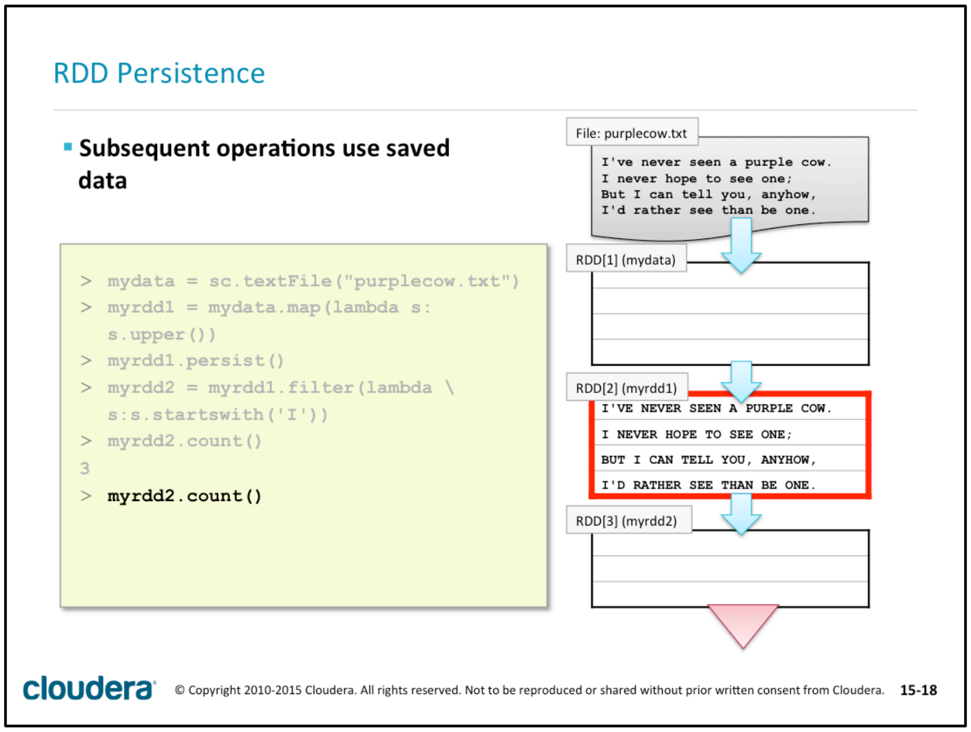
- **Persisting an RDD saves the data (by default in memory)**

```
> mydata = sc.textFile("purplecow.txt")
> myrdd1 = mydata.map(lambda s:
  s.upper())
> myrdd1.persist()
> myrdd2 = myrdd1.filter(lambda \
  s:s.startswith('I'))
> myrdd2.count()
3
```



File: purplecow.txt
I've never seen a purple cow.
I never hope to see one;
But I can tell you, anyhow,
I'd rather see than be one.
RDD[1] (mydata)
I've never seen a purple cow.
I never hope to see one;
But I can tell you, anyhow,
I'd rather see than be one.
RDD[2] (myrdd1)
I'VE NEVER SEEN A PURPLE COW.
I NEVER HOPE TO SEE ONE;
BUT I CAN TELL YOU, ANYHOW,
I'D RATHER SEE THAN BE ONE.
RDD[3] (myrdd2)
I'VE NEVER SEEN A PURPLE COW.
I NEVER HOPE TO SEE ONE;
I'D RATHER SEE THAN BE ONE.

Not we call count, which executes the lineage…and persists RDD2 at the same time.



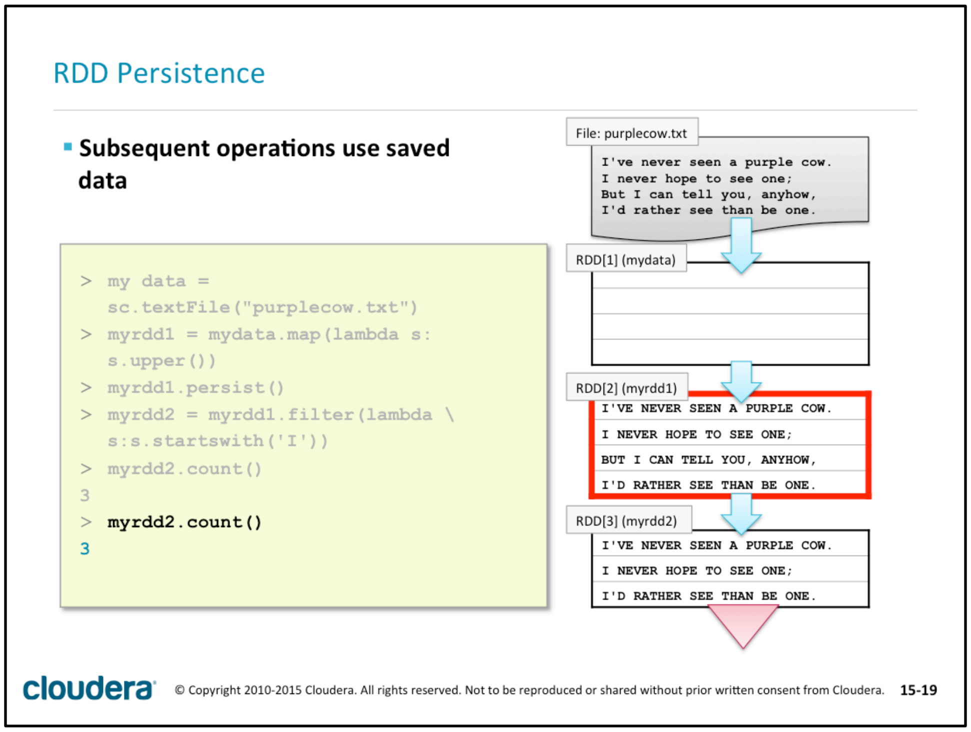
RDD Persistence

- **Subsequent operations use saved data**

```
> mydata = sc.textFile("purplecow.txt")
> myrdd1 = mydata.map(lambda s:
  s.upper())
> myrdd1.persist()
> myrdd2 = myrdd1.filter(lambda \
  s:s.startswith('I'))
> myrdd2.count()
3
> myrdd2.count()
```

File: purplecow.txt
I've never seen a purple cow.
I never hope to see one;
But I can tell you, anyhow,
I'd rather see than be one.
RDD[1] (mydata)
RDD[2] (myrdd1)
I'VE NEVER SEEN A PURPLE COW.
I NEVER HOPE TO SEE ONE;
BUT I CAN TELL YOU, ANYHOW,
I'D RATHER SEE THAN BE ONE.
RDD[3] (myrdd2)

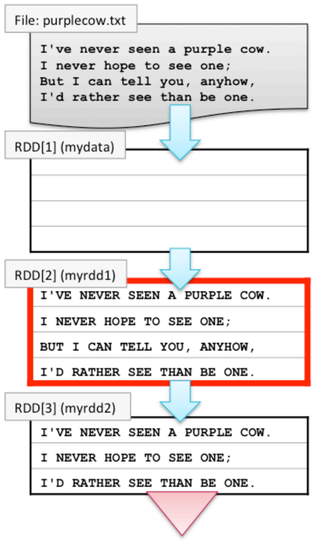
When the computation is complete and the result of count() has been returned, the data is cleared from memory except the cached RDD, which is stored in memory.



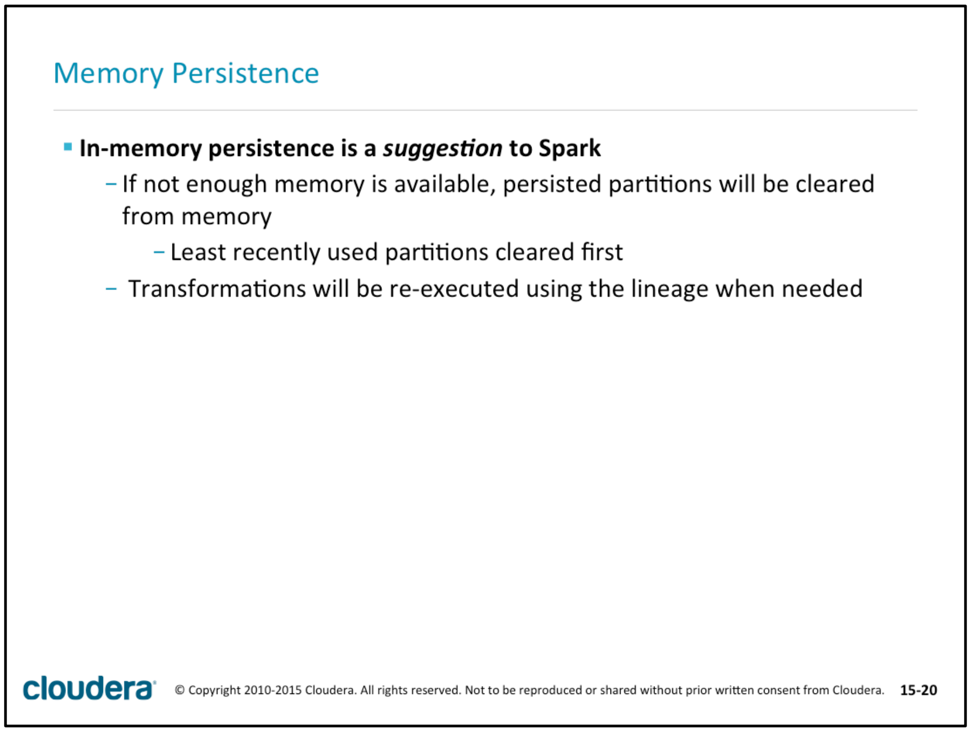
RDD Persistence

- **Subsequent operations use saved data**

```
> my data =
  sc.textFile("purplecow.txt")
> myrdd1 = mydata.map(lambda s:
  s.upper())
> myrdd1.persist()
> myrdd2 = myrdd1.filter(lambda \
  s:s.startswith('I'))
> myrdd2.count()
3
> myrdd2.count()
3
```


File: purplecow.txt
I've never seen a purple cow.
I never hope to see one;
But I can tell you, anyhow,
I'd rather see than be one.
RDD[1] (mydata)
RDD[2] (myrdd1)
I'VE NEVER SEEN A PURPLE COW.
I NEVER HOPE TO SEE ONE;
BUT I CAN TELL YOU, ANYHOW,
I'D RATHER SEE THAN BE ONE.
RDD[3] (myrdd2)
I'VE NEVER SEEN A PURPLE COW.
I NEVER HOPE TO SEE ONE;
I'D RATHER SEE THAN BE ONE.

So when we call count, it executes the lineage of RDD3…which depends on RDD2. But because RDD2 is persisted, it uses that cached data, and does not need to execute any further back in the lineage.



Memory Persistence

- **In-memory persistence is a *suggestion* to Spark**
  - If not enough memory is available, persisted partitions will be cleared from memory
    - Least recently used partitions cleared first
  - Transformations will be re-executed using the lineage when needed

Important point here. Short slide, but spend some time here anyway. Much confusion of this point. "What happens if there's not enough memory?" it will store as much of the RDD as it can in memory, and compute the rest when needed using the lineage. Will the data spill to disk, like MapReduce? No, not by default.

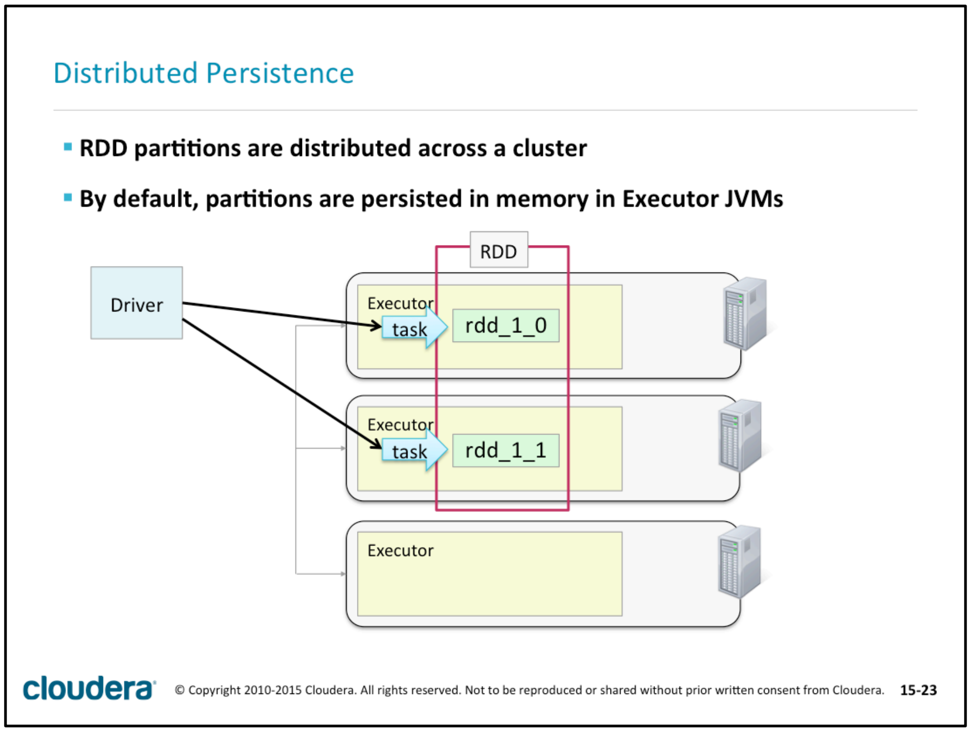
Chapter Topics

Spark RDD Persistence	Distributed Data Processing with Spark

- RDD Lineage
- RDD Persistence Overview
- **Distributed Persistence**
- Conclusion
- Homework: Persist an RDD

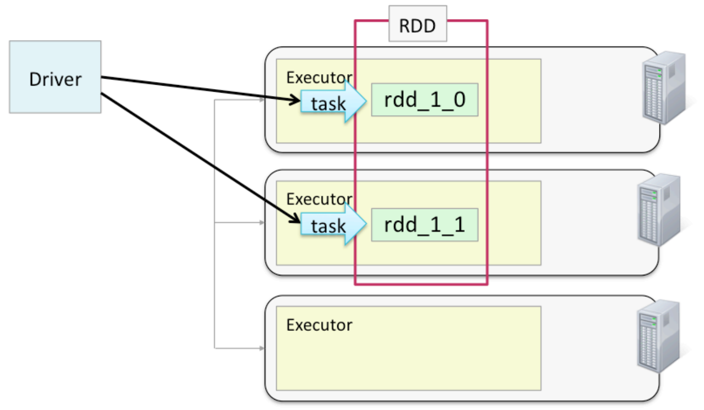
Persistence and Fault-Tolerance

- **RDD = *Resilient* Distributed Dataset**
  - Resiliency is a product of tracking lineage
  - RDDs can always be recomputed from their base if needed

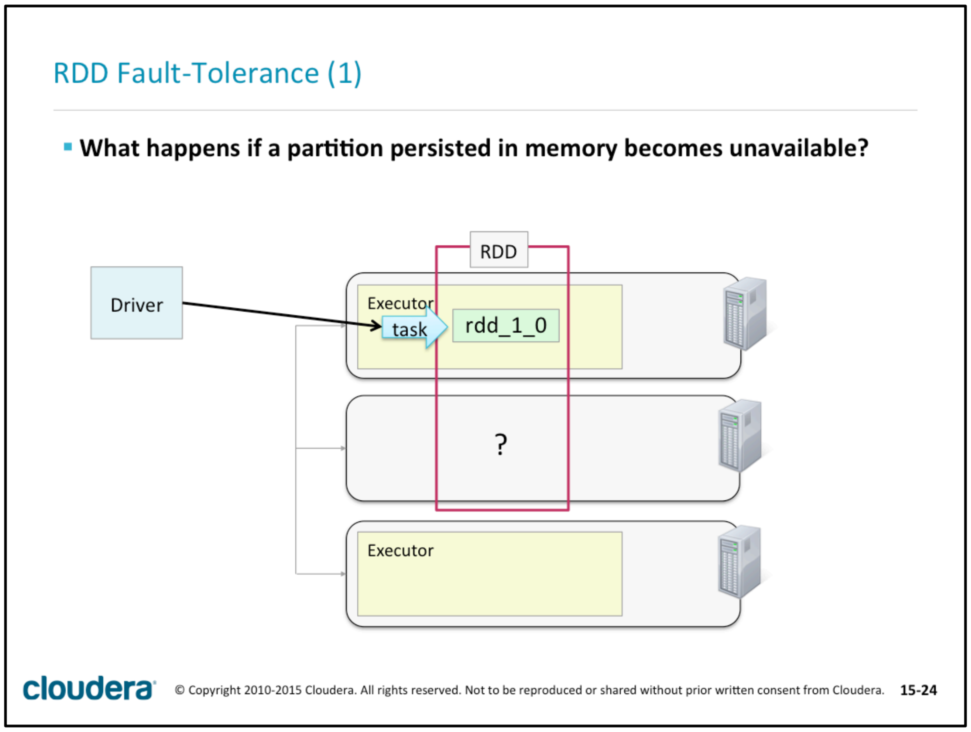


Distributed Persistence

- RDD partitions are distributed across a cluster
- By default, partitions are persisted in memory in Executor JVMs

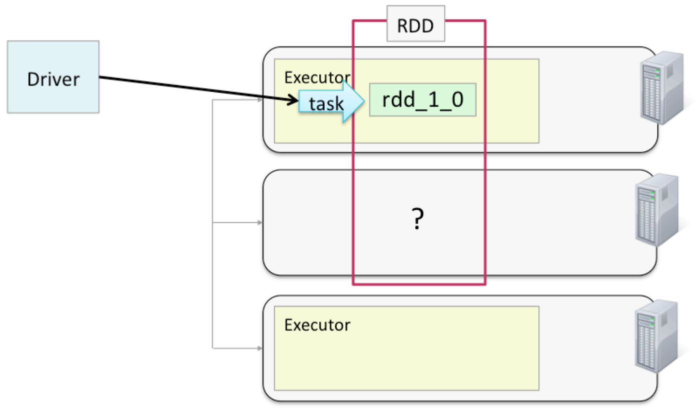


START BUILD
Here's an RDD that has two partitions, cached on nodes 1 and 2.

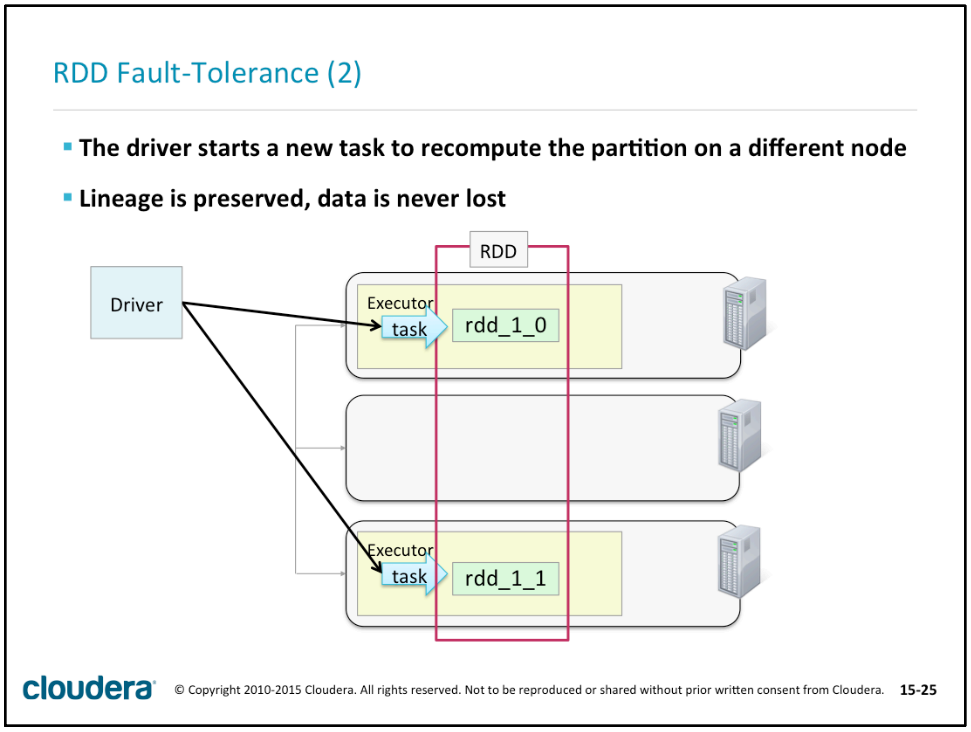


RDD Fault-Tolerance (1)

- **What happens if a partition persisted in memory becomes unavailable?**

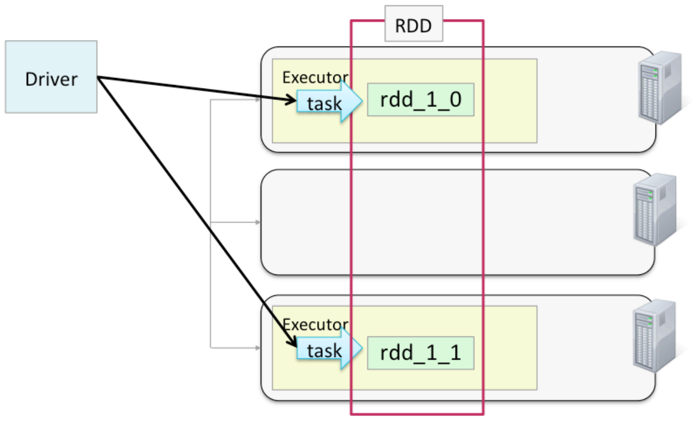


But node 2 crashes, and the client is trying to get that RDD’s value.

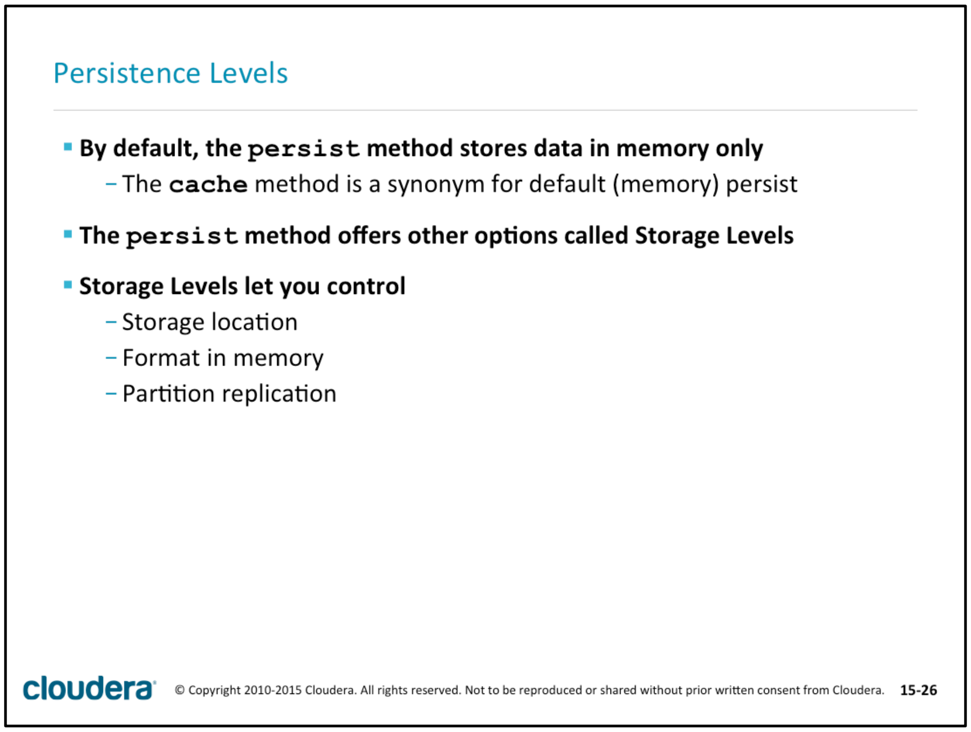


RDD Fault-Tolerance (2)

- The driver starts a new task to recompute the partition on a different node
- Lineage is preserved, data is never lost



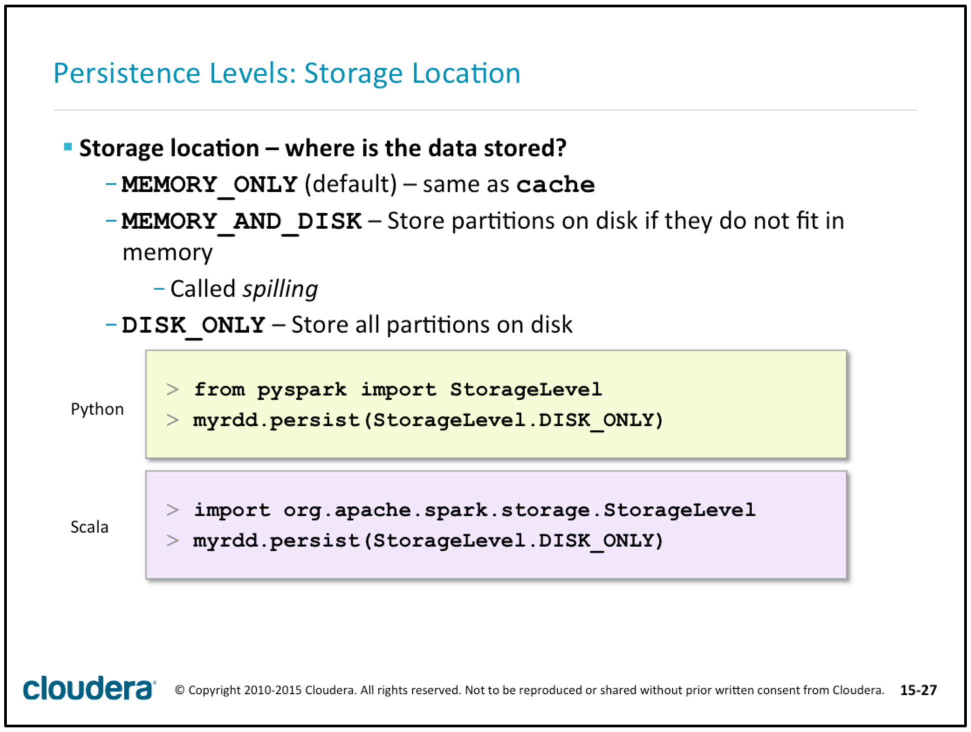
END BUILD



Persistence Levels

- **By default, the `persist` method stores data in memory only**
  - The **`cache`** method is a synonym for default (memory) persist
- **The `persist` method offers other options called Storage Levels**
- **Storage Levels let you control**
  - Storage location
  - Format in memory
  - Partition replication

Replication: You can actually define your own StorageLevel for replication greater than 2 if you wish


Persistence Levels: Storage Location

- **Storage location – where is the data stored?**
  - **`MEMORY_ONLY`** (default) – same as **`cache`**
  - **`MEMORY_AND_DISK`** – Store partitions on disk if they do not fit in memory
    - Called *spilling*
  - **`DISK_ONLY`** – Store all partitions on disk

Python

```
> from pyspark import StorageLevel
> myrdd.persist(StorageLevel.DISK_ONLY)
```

Scala

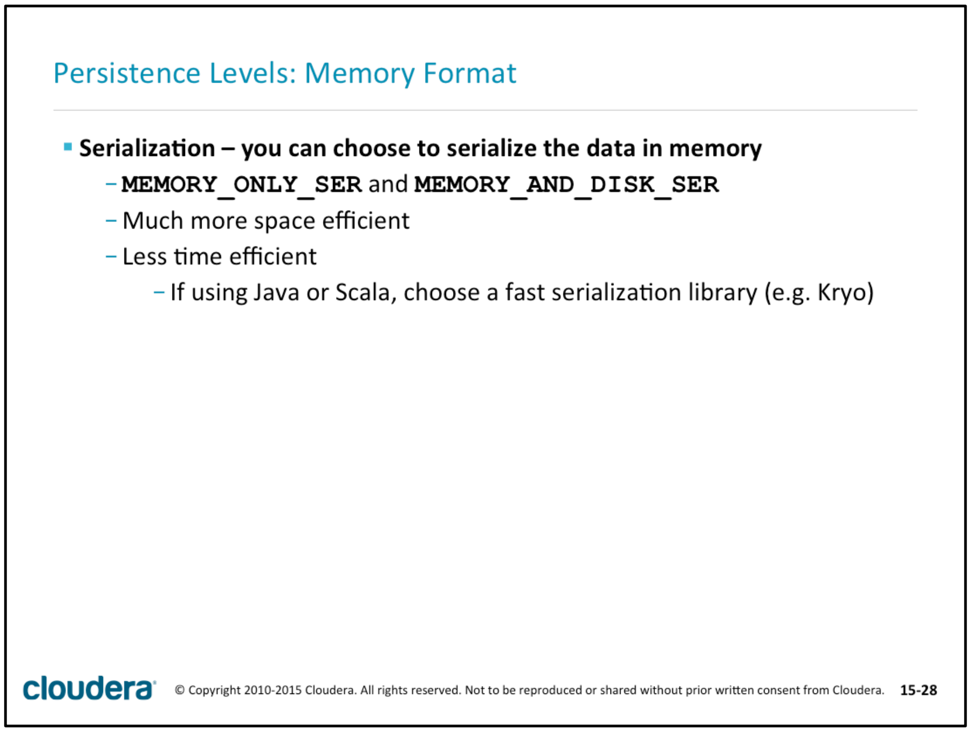
```
> import org.apache.spark.storage.StorageLevel
> myrdd.persist(StorageLevel.DISK_ONLY)
```

http://spark.apache.org/docs/latest/scala-programming-guide.html#rdd-persistence
http://spark.apache.org/docs/latest/api/core/
index.html#org.apache.spark.storage.StorageLevel

Predefined Storage Levels are defined on the StorageLevels interface, which you need to import to use.

From the docs: there's also an "experimental" storage level:
`OFF_HEAP` (experimental): Store RDD in serialized format in Tachyon. Compared to MEMORY_ONLY_SER, OFF_HEAP reduces garbage collection overhead and allows executors to be smaller and to share a pool of memory, making it attractive in environments with large heaps or multiple concurrent applications. Furthermore, as the RDDs reside in Tachyon, the crash of an executor does not lead to losing the in-memory cache. In this mode, the memory in Tachyon is discardable. Thus, Tachyon does not attempt to reconstruct a block that it evicts from memory.
We don't cover Tachyon in class so there's no reason to mention this, unless a student specifically asks about it.

When an RDD cache level is MEMORY_AND_DISK it means each of the RDD's partitions are either completely saved to disk or completely persisted in RAM. If a



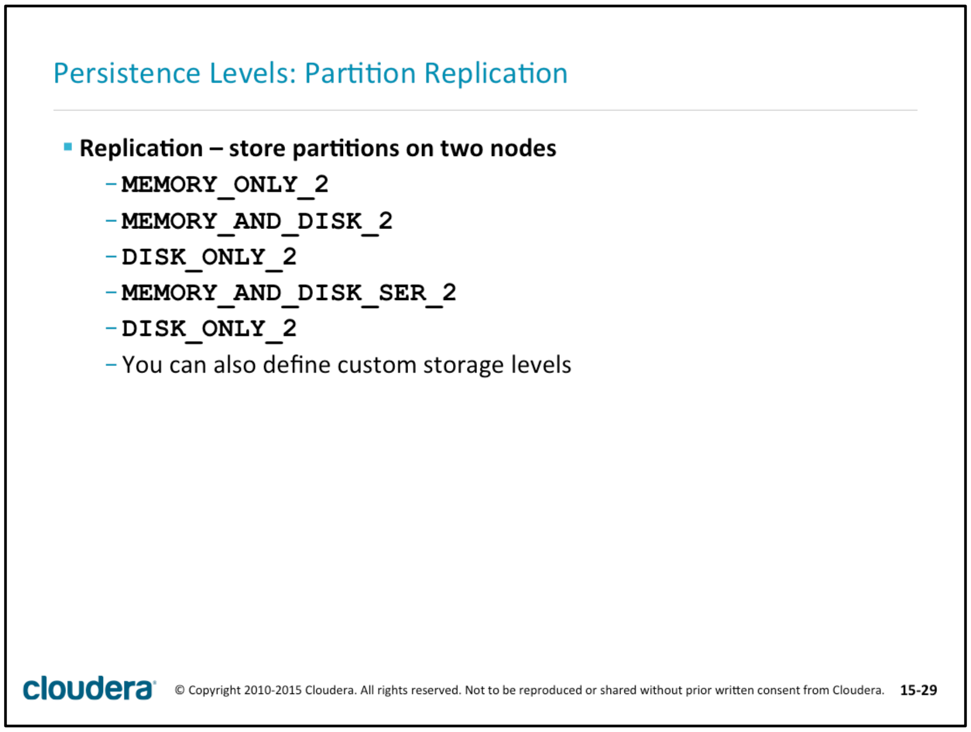
Persistence Levels: Memory Format

- **Serialization – you can choose to serialize the data in memory**
  - **`MEMORY_ONLY_SER`** and **`MEMORY_AND_DISK_SER`**
  - Much more space efficient
  - Less time efficient
    - If using Java or Scala, choose a fast serialization library (e.g. Kryo)

Kryo is an alterative to Java/Scala built-in serialization. Set using the spark.serializer property of an application, e.g.

```
spark.serializer
org.apache.spark.serializer.KryoSerializer
```

This will work with all built-in Java/Scala classes and Spark classes. Custom classes need to be registered with Kryo, which is outside scope for this course.
This is not relevant for Python – Python serialization is done using pickle which is quite efficient.



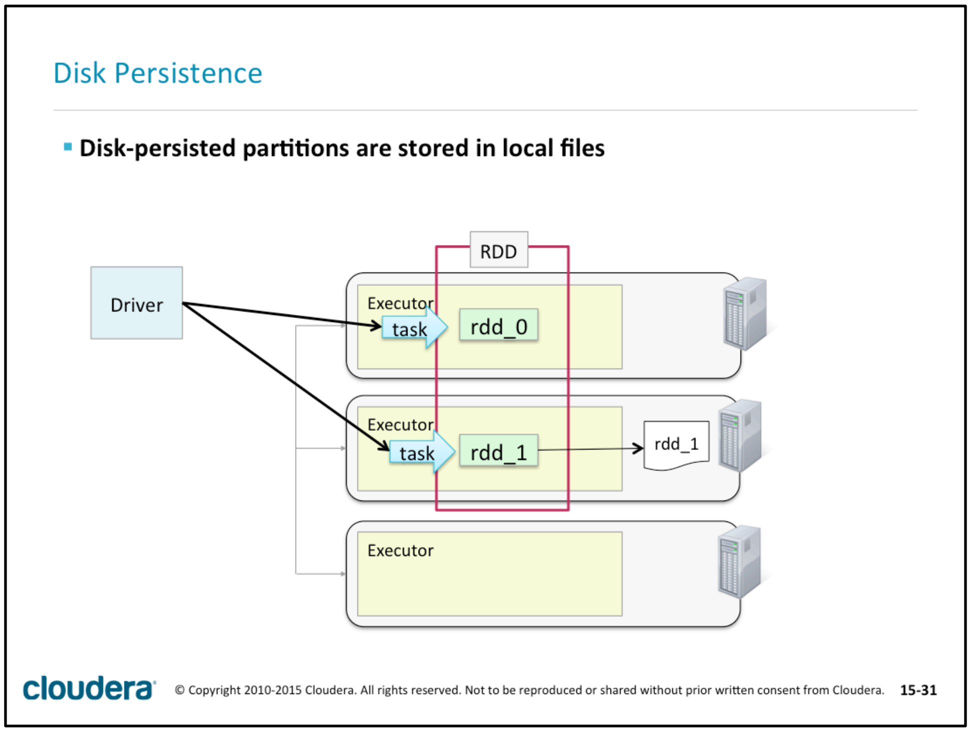
Persistence Levels: Partition Replication

- **Replication – store partitions on two nodes**
  - **`MEMORY_ONLY_2`**
  - **`MEMORY_AND_DISK_2`**
  - **`DISK_ONLY_2`**
  - **`MEMORY_AND_DISK_SER_2`**
  - **`DISK_ONLY_2`**
  - You can also define custom storage levels

Replication: You can actually define your own StorageLevel for replication greater than 2 if you wish

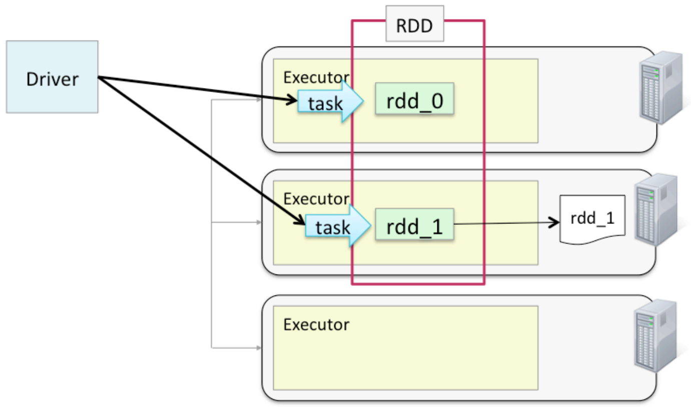
Changing Persistence Options

- **To stop persisting and remove from memory and disk**
  - **`rdd.unpersist()`**
- **To change an RDD to a different persistence level**
  - Unpersist first



Disk Persistence

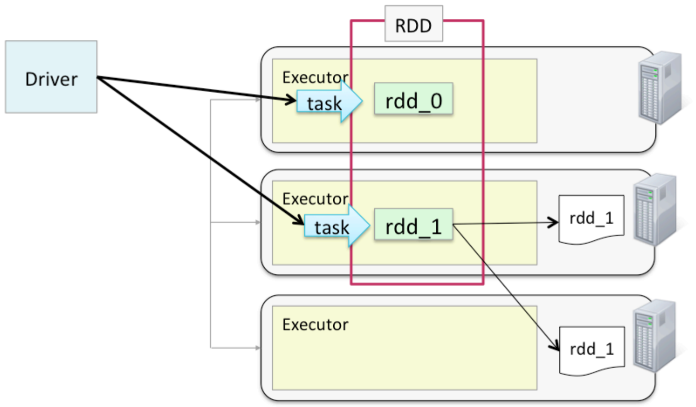
- **Disk-persisted partitions are stored in local files**

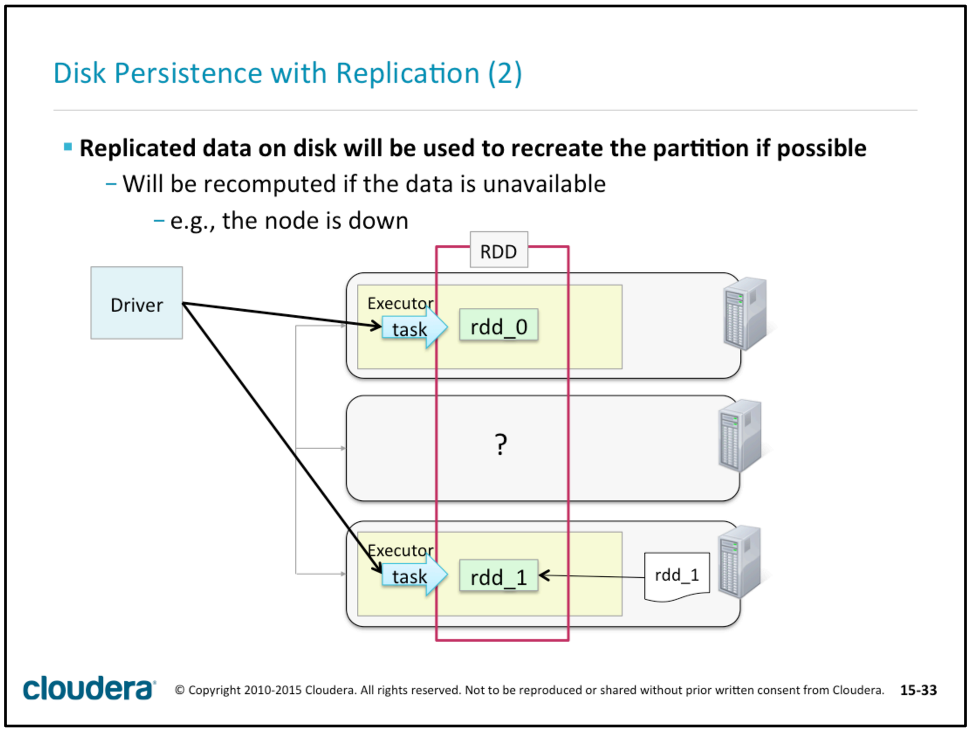


START BUILD

Disk Persistence with Replication (1)

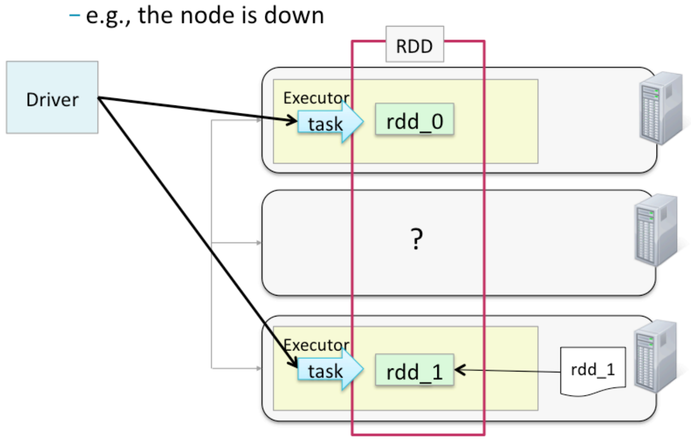
- **Persistence replication makes recomputation less likely to be necessary**





Disk Persistence with Replication (2)

- **Replicated data on disk will be used to recreate the partition if possible**
  - Will be recomputed if the data is unavailable
    - e.g., the node is down



END BUILD
This example shows disk replication, but the same logic applies to in memory replication. If a node goes down, another node already has the data cached on it.

note that the cost of replication isn't just memory or disk space, but also network bandwidth.

When and Where to Persist

- **When should you persist a dataset?**
  - When a dataset is likely to be re-used
    - e.g., iterative algorithms, machine learning
- **How to choose a persistence level**
  - Memory only – when possible, best performance
    - Save space by saving as serialized objects in memory if necessary
  - Disk – choose when recomputation is more expensive than disk read
    - e.g., expensive functions or filtering large datasets
  - Replication – choose when recomputation is more expensive than memory

Chapter Topics

Spark RDD Persistence | **Distributed Data Processing with Spark**

- RDD Lineage
- RDD Persistence Overview
- Distributed Persistence
- **Conclusion**
- Homework: Persist an RDD

Essential Points

- **Spark keeps track of each RDD's lineage**
  - Provides fault tolerance
- **By default, every RDD operation executes the entire lineage**
- **If an RDD will be used multiple times, persist it to avoid re-computation**
- **Persistence options**
  - Location – memory only, memory and disk , disk only
  - Format – in-memory data can be serialized to save memory (but at the cost of performance)
  - Replication – saves data on multiple nodes in case a node goes down, for job recovery without recomputation

Chapter Topics

Spark RDD Persistence | **Distributed Data Processing with Spark**

- RDD Lineage
- RDD Persistence Overview
- Distributed Persistence
- Conclusion
- **Homework: Persist an RDD**

Homework: Persist an RDD

- **In this homework assignment you will**
  - Persist an RDD before reusing it
  - Use the Spark Application UI to see how an RDD is persisted
- **Please refer to the Homework description**