Spark Azure Databricks

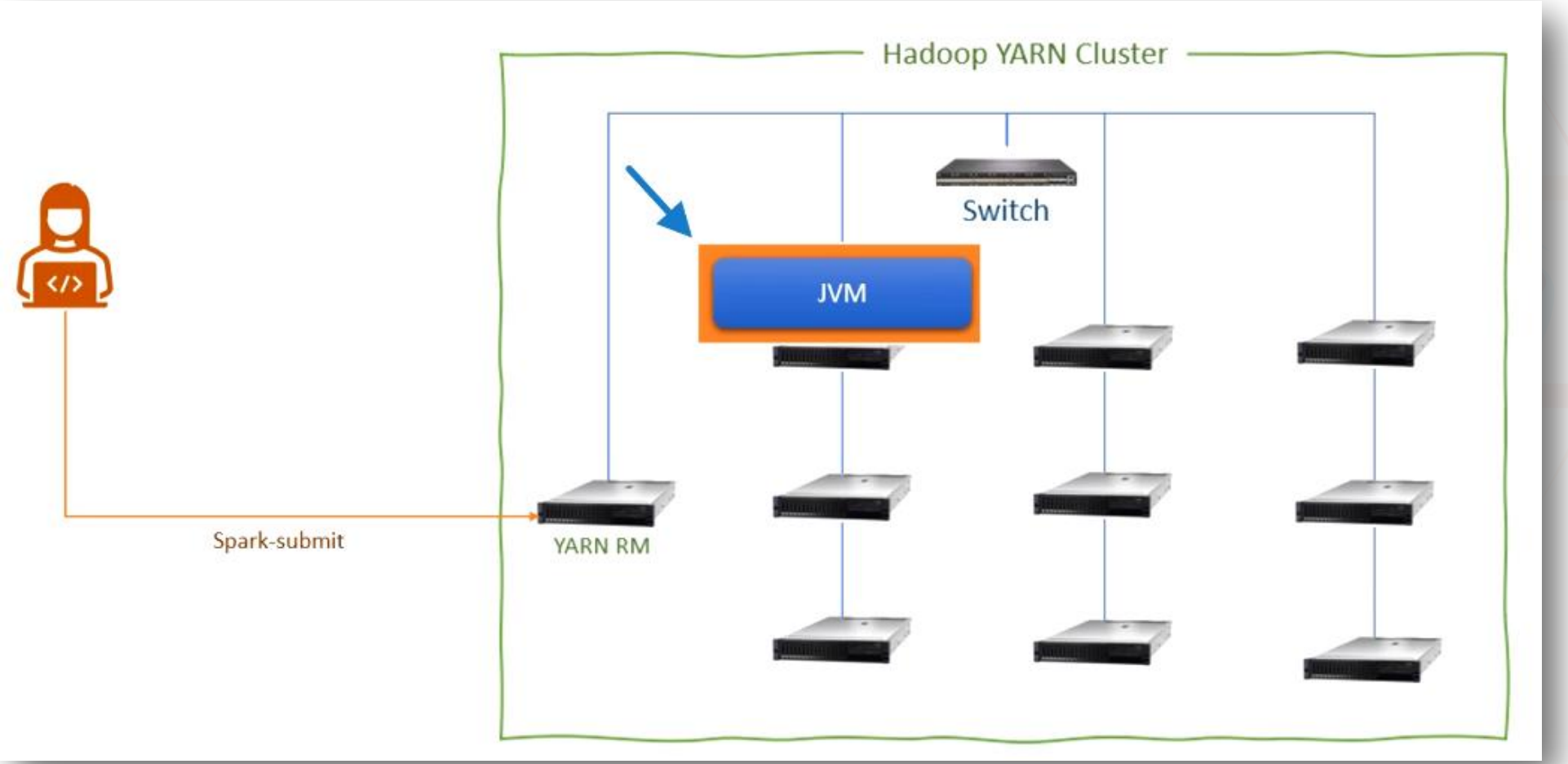
Databricks Spark Certification and Beyond

Module:
Advanced Spark

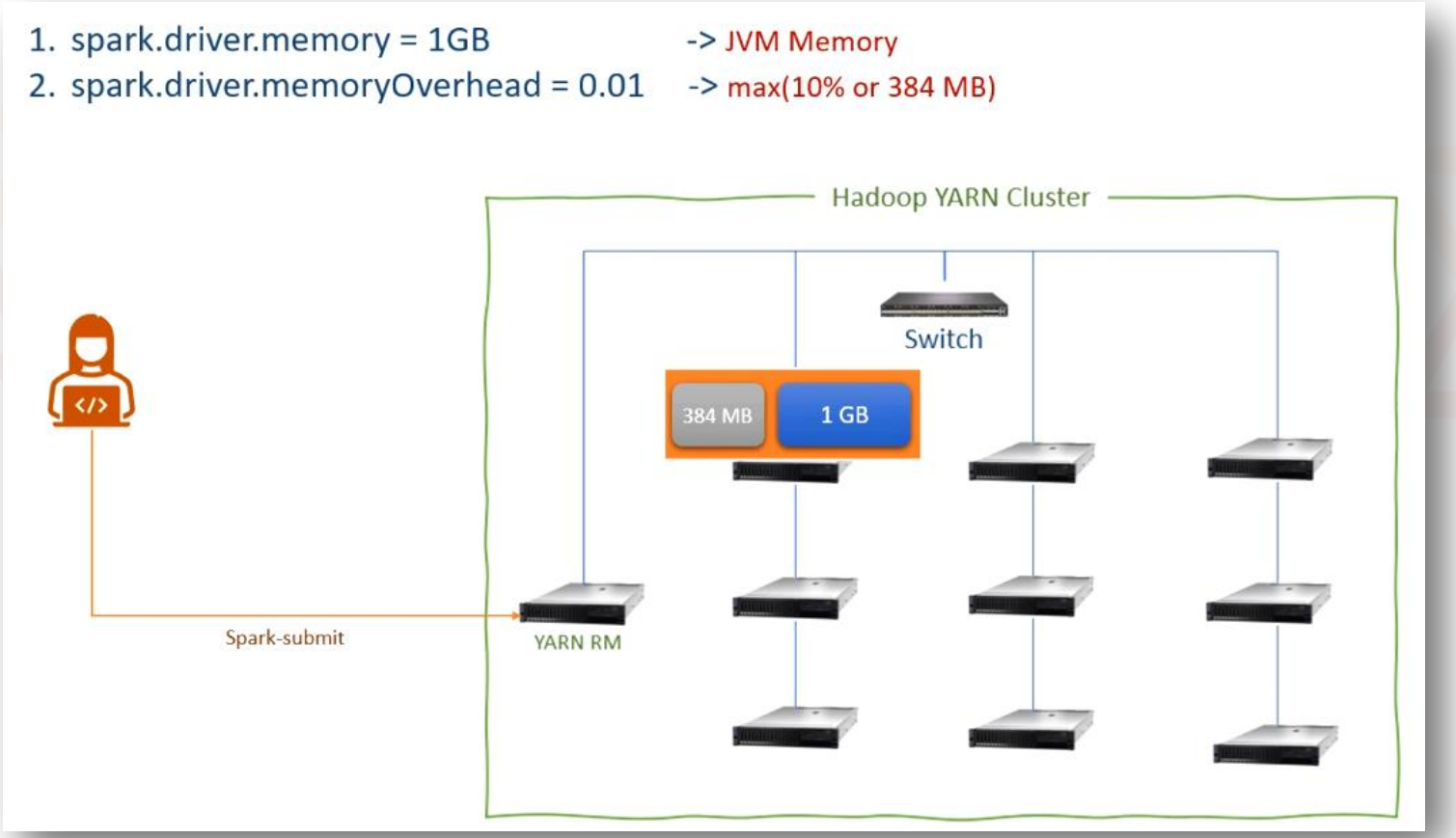
Lecture:
Memory Allocation

Spark Memory Allocation

In this video, I will talk about the memory allocation for Spark driver and Spark executor.
Assume you submitted a spark application in a YARN cluster.
The YARN RM will allocate an application master (AM) container and start the driver JVM in the container. The driver will start with some memory allocation which you requested.



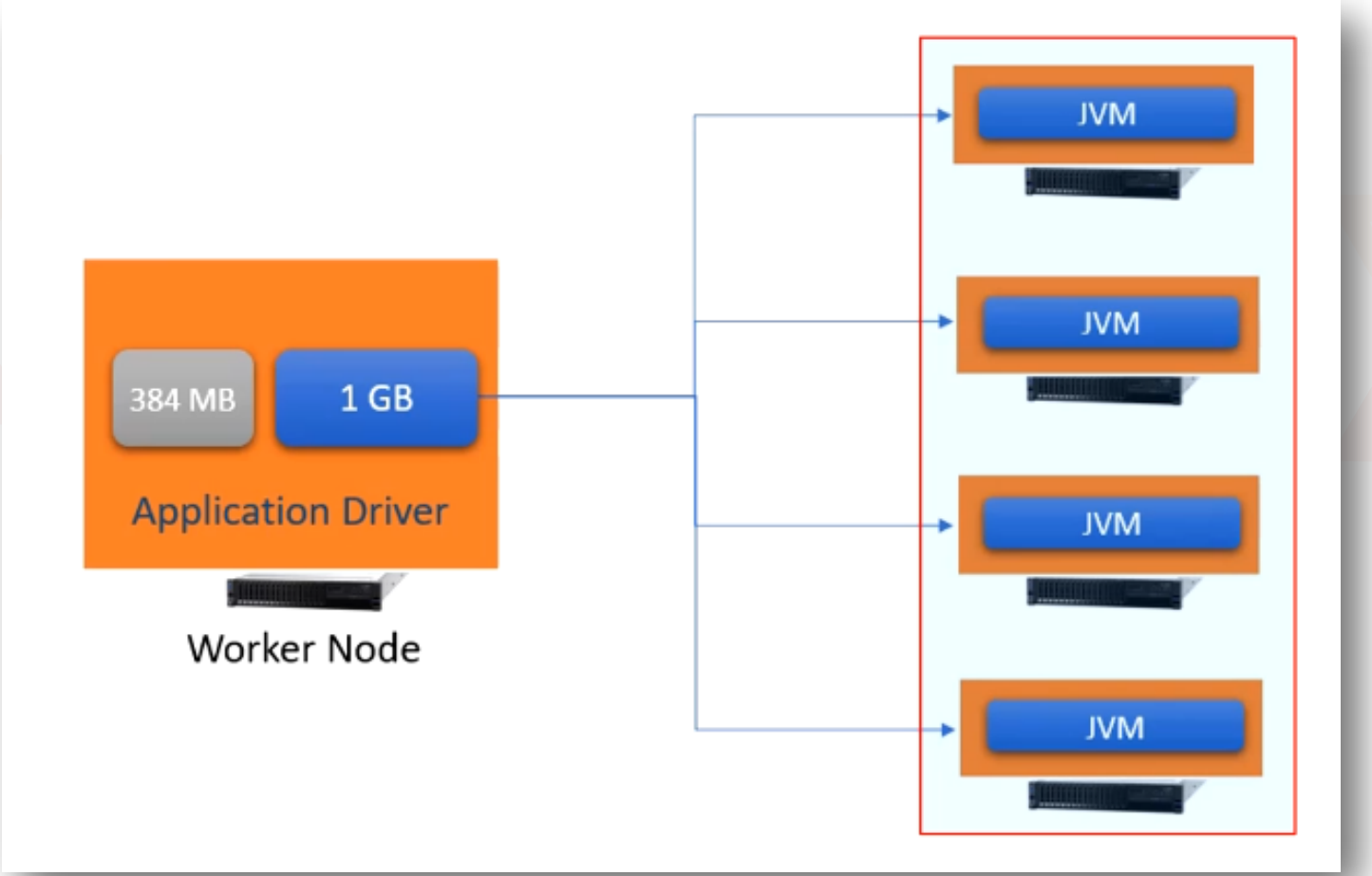
Do you know how to ask for the driver's memory? You can ask for the driver memory using two configurations: ***spark.driver.memory*** and ***spark.driver.memoryOverhead***
So let's assume you asked for the spark.driver.memory = 1GB.
And the default value of spark.driver.memoryOverhead = 0.10, that means it will allocate 10% of your requested memory or 384 MB, whatever is higher for container overhead.



The YARN RM will allocate 1 GB of memory for the driver JVM. And it will also allocate 10% of your requested memory or 384 MB, whatever is higher for container overhead.

We asked for 1 GB *spark.driver.memory*. So the 10% of 1 GB is 100 MB. But 100 MB is less than the 384 MB. So the YARN RM will allocate 384 MB for overhead.
So what is the total memory for the container? It comes to 1 GB + 384 MB.
But what is the purpose of 384 MB overhead? The overhead memory is used by the container process or any other non JVM process within the container. Your Spark driver uses all the JVM heap but nothing from the overhead.
And that's all about the driver memory allocation.

Now the driver is started with 1 GB of JVM heap. Right?
So the driver will again request for the executor containers from the YARN.
The YARN RM will allocate a bunch of executor containers.
But how much memory do you get for each executor container?



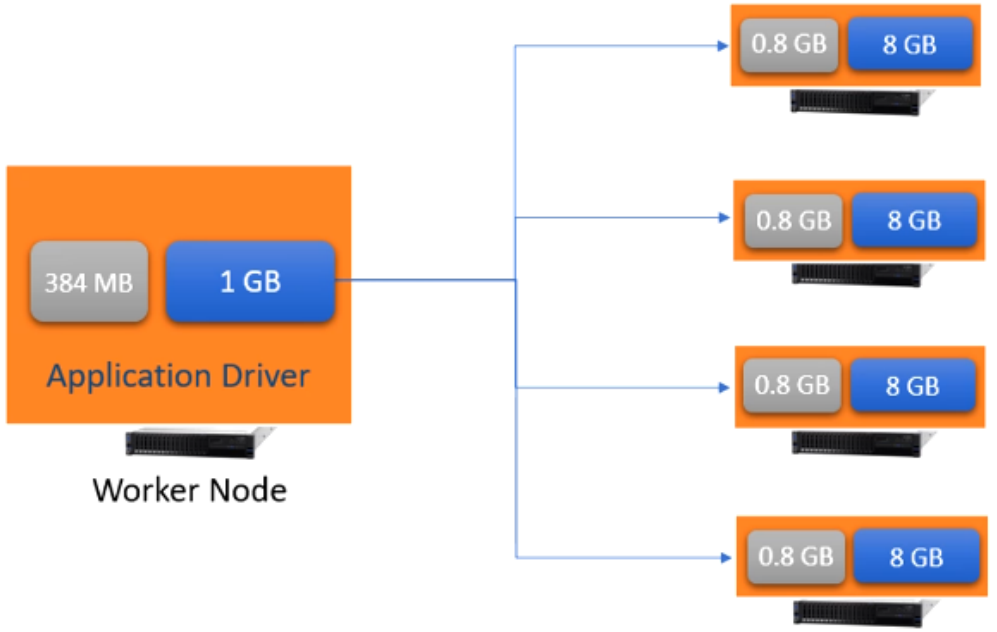
The total memory allocated to the executor container, is the sum of the following:

1. Overhead Memory - *spark.executor.memoryOverhead*
2. Heap Memory - *spark.executor.memory*
3. Off Heap Memory - *spark.memory.offHeap.size*
4. PySpark Memory - *spark.executor.pyspark.memory*

So a Spark driver will ask for executor container memory using the above four configurations. The driver will look at all these configurations to calculate your memory requirement and sum it up.

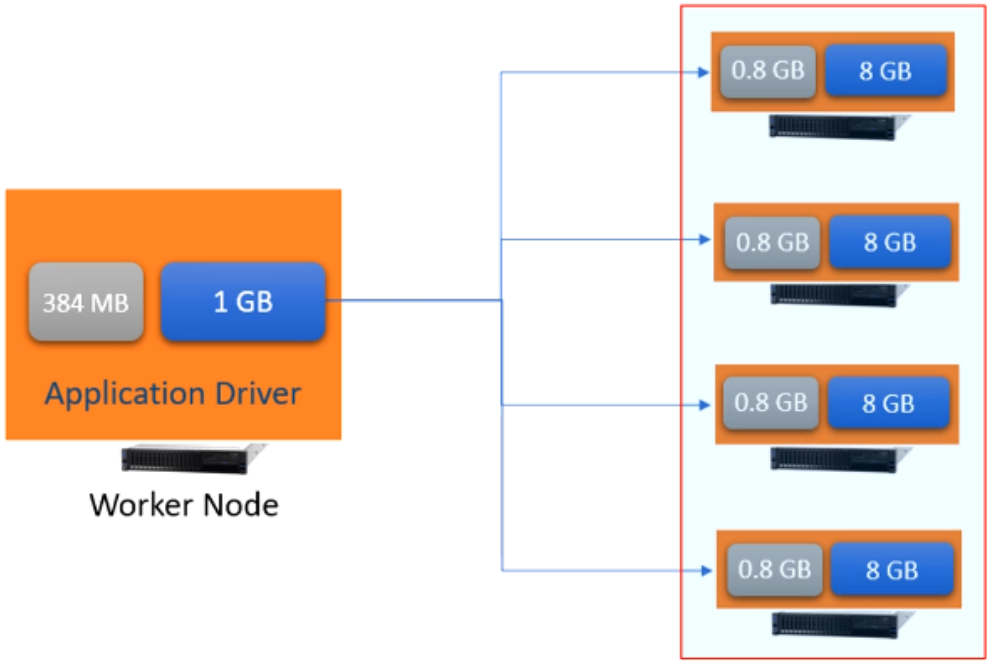
Now let's assume you asked for *spark.executor.memory = 8 GB*
The default value of *spark.executor.memoryOverhead = 10%*
Let's assume the other two configurations are not set, and the default value is zero.
So how much memory do you get for your executor container?

1. Overhead Memory -> spark.executor.memoryOverhead = 0.1
2. Heap Memory -> spark.executor.memory = 8GB
3. Off Heap Memory -> spark.memory.offHeap.size = 0
4. PySpark Memory -> spark.executor.pyspark.memory = 0



You asked *spark.executor.memory = 8 GB*, so you will get 8 GB for JVM.
Then you asked for *spark.executor.memoryOverhead = 10%,* so you will get 800 MB extra for the overhead. And the total container memory comes to 8800 MB.
So the driver will ask for 8.8 GB containers to the YARN RM.

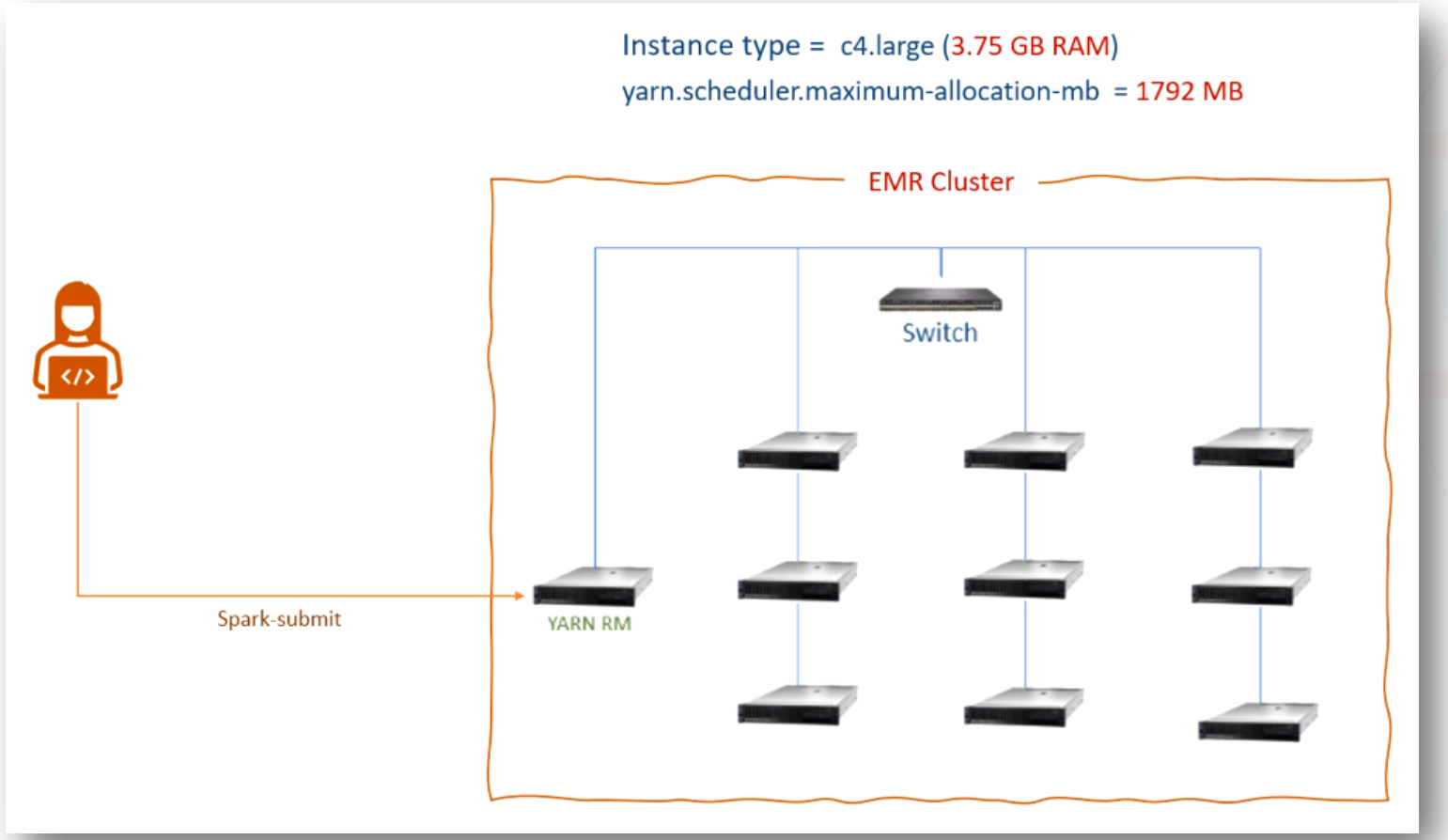
1. Overhead Memory -> spark.executor.memoryOverhead = 0.1
2. Heap Memory -> spark.executor.memory = 8GB
3. Off Heap Memory -> spark.memory.offHeap.size = 0
4. PySpark Memory -> spark.executor.pyspark.memory = 0



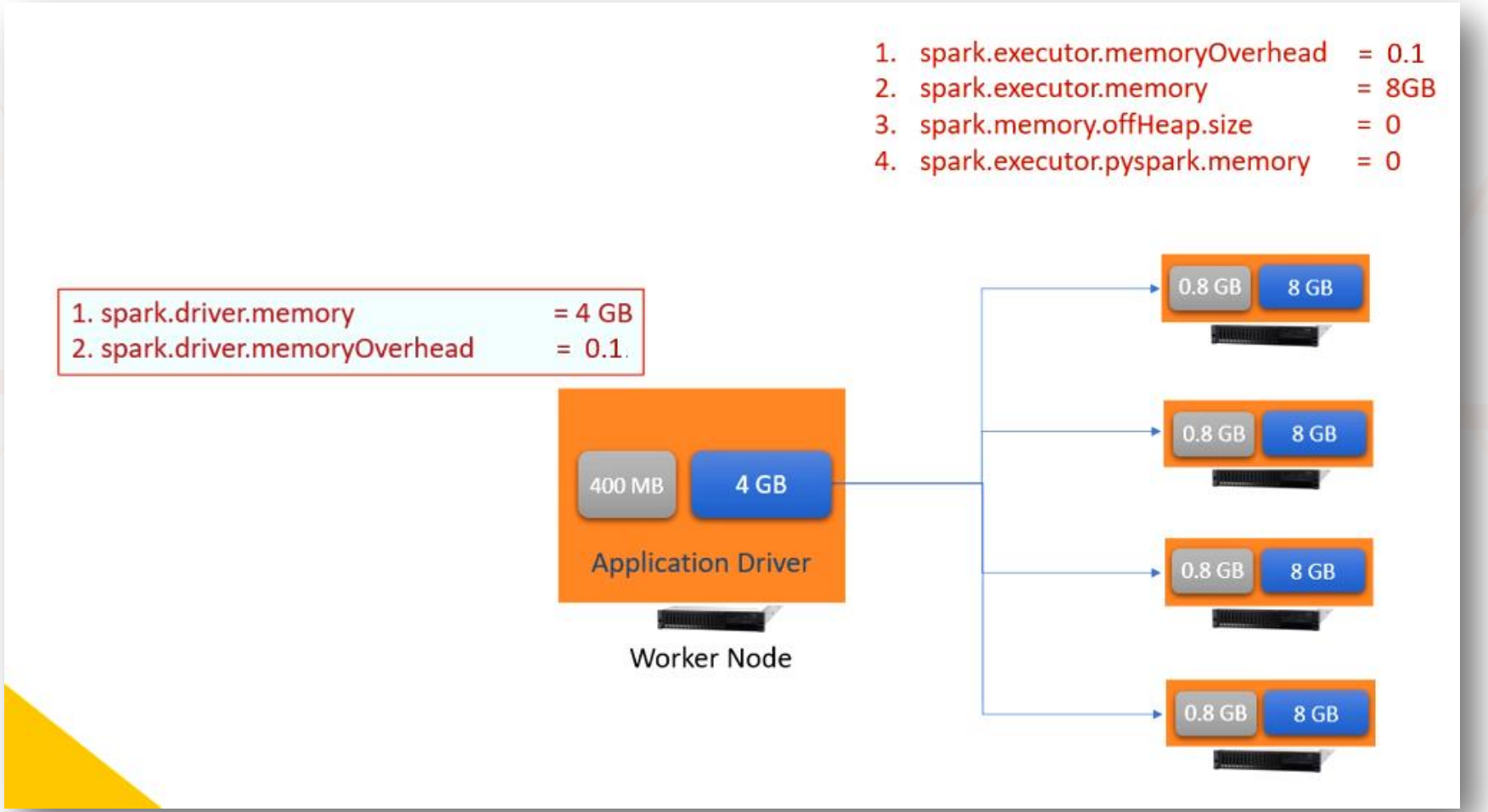
But do we get an 8.8 GB container? Well, that depends on your cluster configuration.
The container should run on a worker node in the YARN cluster.
What if the worker node is a 6 GB machine? YARN cannot allocate an 8 GB container on a 6 GB machine. Because there is not enough physical memory, before you ask for the driver or executor memory, you should check with your cluster admin for the maximum allowed value.
If you are using YARN RM, you should look for the following configurations:

1. *yarn.scheduler.maximum-allocation-mb*
2. *yarn.nodemanager.resource.memory-mb*

For example, let's assume you are using AWS c4.large instance type to set up your Spark cluster using AWS EMR. The c4.large instance comes with 3.75 GB RAM. You launched an EMR cluster using c4.large instances. The EMR cluster will start with the default value of *yarn.scheduler.maximum-allocation-mb = 1792.* That means you cannot ask for a container of higher than 1.792 GB. That's your physical memory limit for each container. Even if you ask, you are not going to get it.



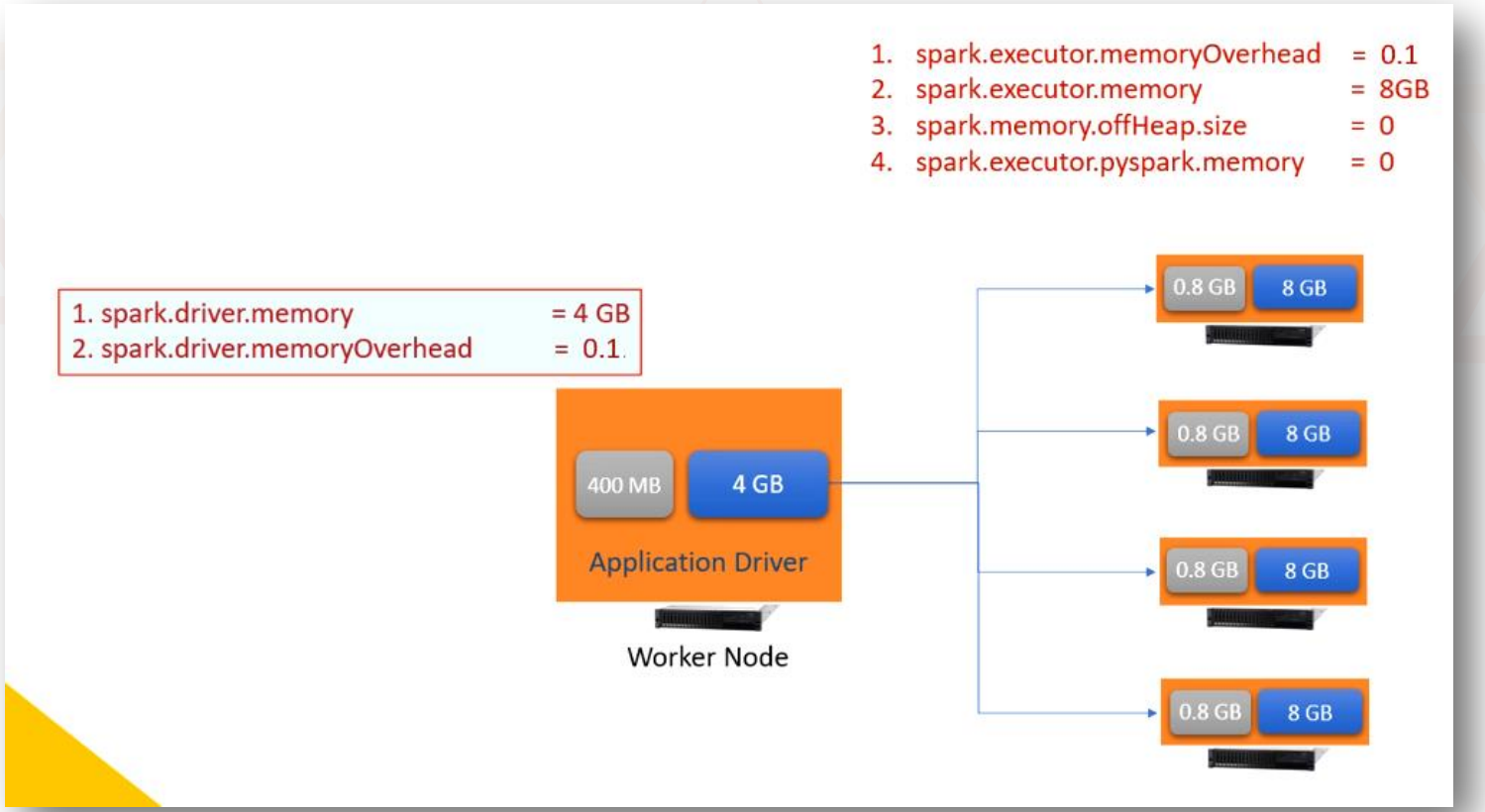
Now let's come back to our scenario. I am assuming that we have enough physical memory on the worker nodes. So we learned about the driver and executor memory.
The total physical memory for a driver container comes from the following two configurations highlighted below. Once allocated, it becomes your physical memory limit for your spark driver.



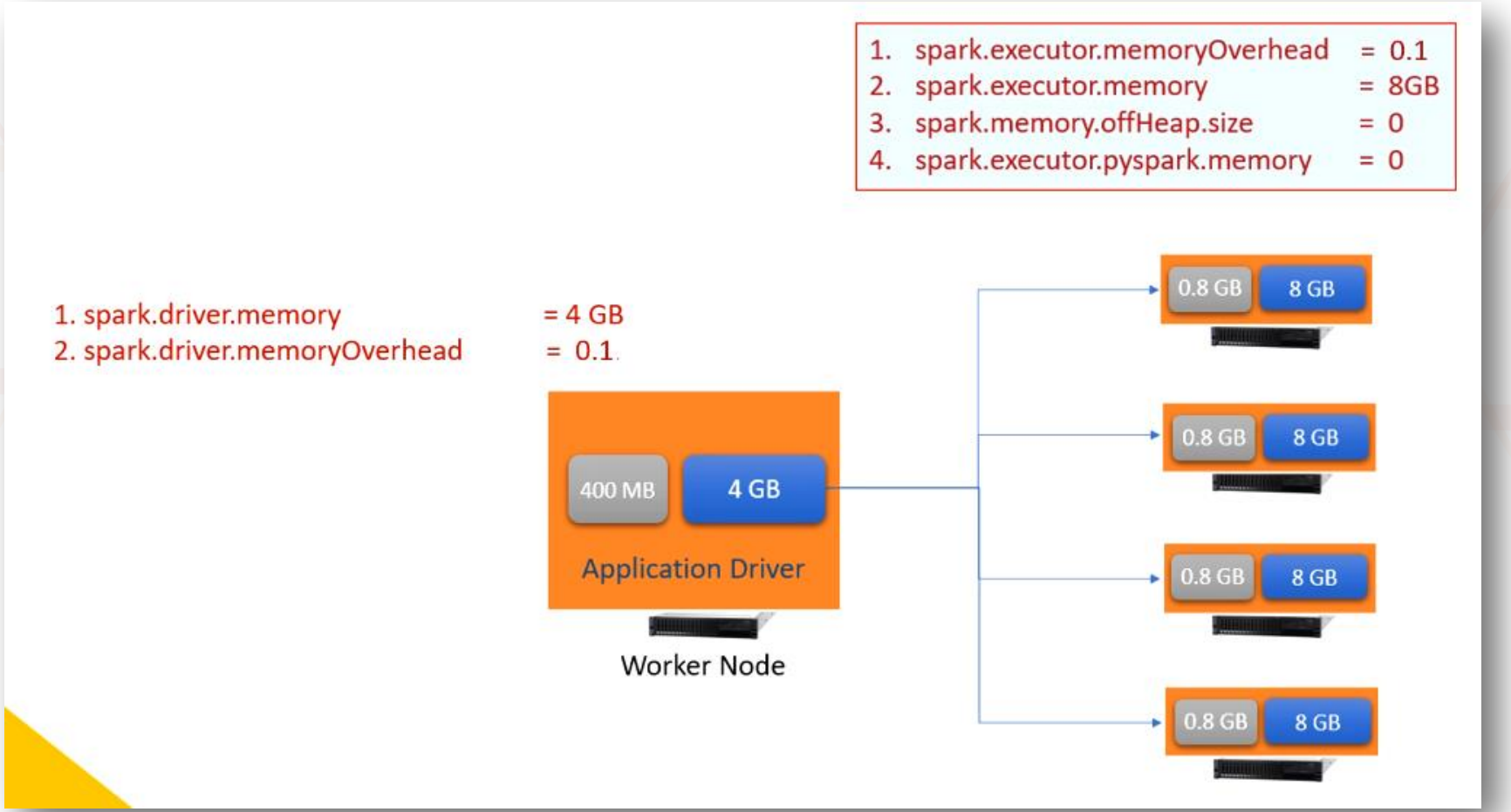
So let us take for example, you asked for a 4 GB *spark.driver.memory*, you will get 4 GB JVM heap and 400 MB off JVM Overhead memory. Now you have three limits:

1. Your Spark driver JVM cannot use more than 4 GB.
2. Your non-JVM workload in the container cannot use more than 400 MB.
3. Your container cannot use more than 4.4 GB of memory in total.

If any of these limits are violated, you will see an OOM exception.



Now let me summarize executor memory allocation.
The total physical memory for an executor container comes from the following four configurations highlighted below. The default value for the third and fourth configurations are not defined. So you can consider them zero.



Now let's assume you asked for *spark.executor.memory = 8 GB*. So you will get 8 GB for the Spark executor JVM. You will also get 800 MB for overhead. The total physical memory of your container is 8.8 GB. Now you have three limits:

1. Your executor JVM cannot use more than 8 GB of memory.
2. Your non JVM processes cannot use more than 800 MB.
3. Your container has a maximum physical limit of 8.8 GB.

You will see an OOM exception when any of these limits are crossed.

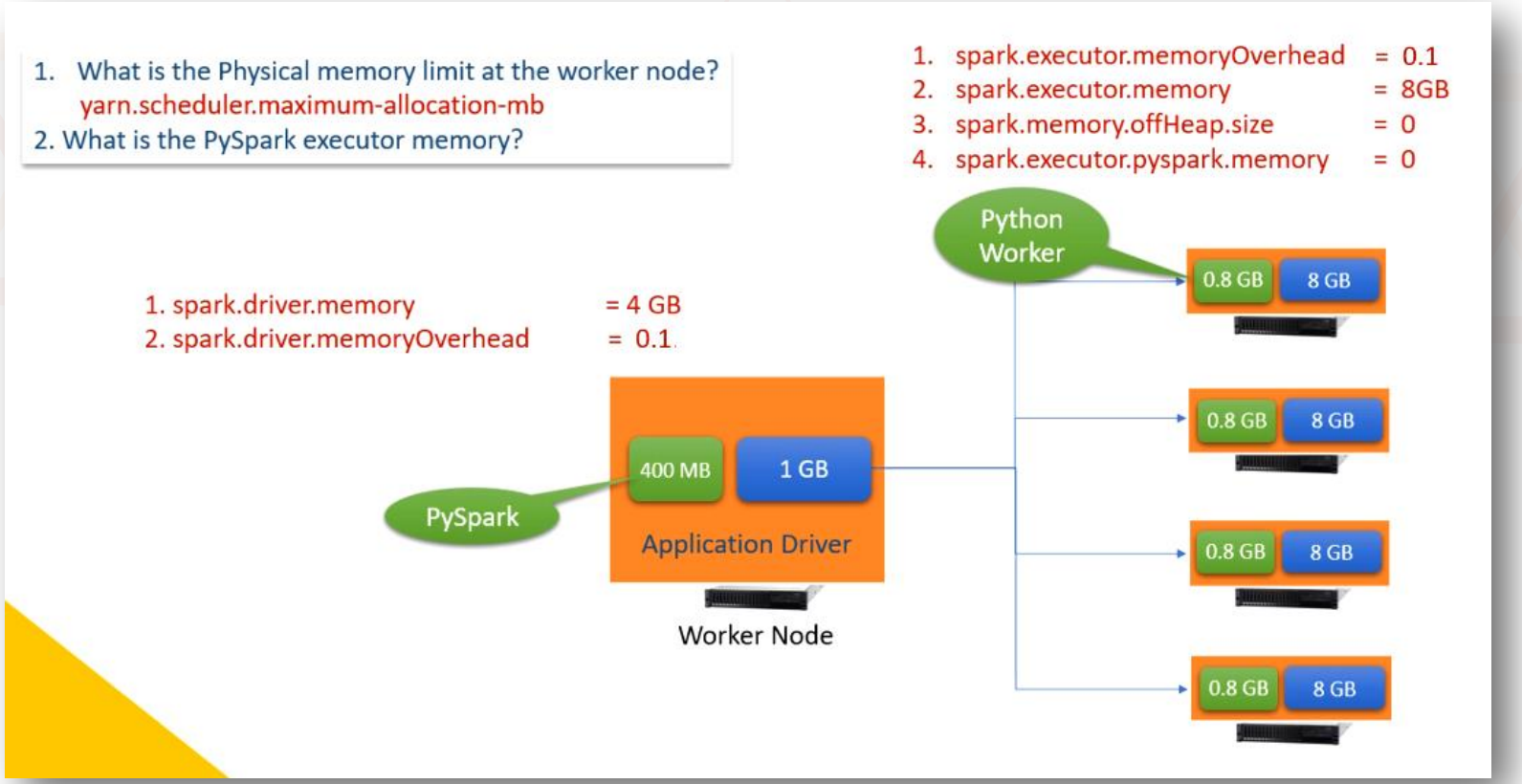
But you should ask two more questions:

1. What is the Physical memory limit at the worker node?
2. What is the PySpark executor memory?

I already talked about the Physical memory limit for your workers.
You should look out for *yarn.scheduler.maximum-allocation-mb* for the maximum limit. You cannot get more than the maximum-allocation-mb.

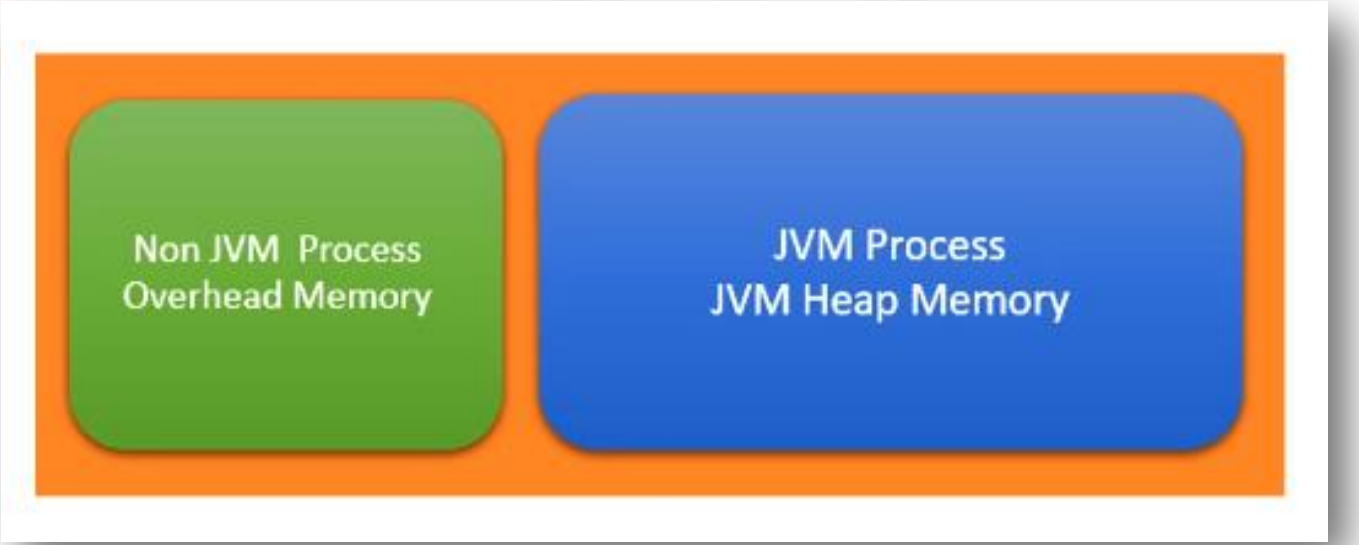
The next question is, how much memory do you get for your PySpark?
We didn't talk about the PySpark memory yet. All we are talking about is JVM and non-JVM memory.
You do not need to worry about PySpark memory if you write your Spark application in Java or Scala. But if you are using PySpark, this question becomes critical.
Let's try to understand.

So for this example, how much memory do you get for PySpark? The answer might surprise you. PySpark is not a JVM process. So you will not get anything from those 8 GBs. All you have is 800 MB of overhead memory. Some 300 to 400 MB of this is constantly consumed by the container processes and other internal processes. So your PySpark will get approximately 400 MB. So now you have one more limit. If your PySpark consumes more than what can be accommodated in the overhead, you will see an OOM error.

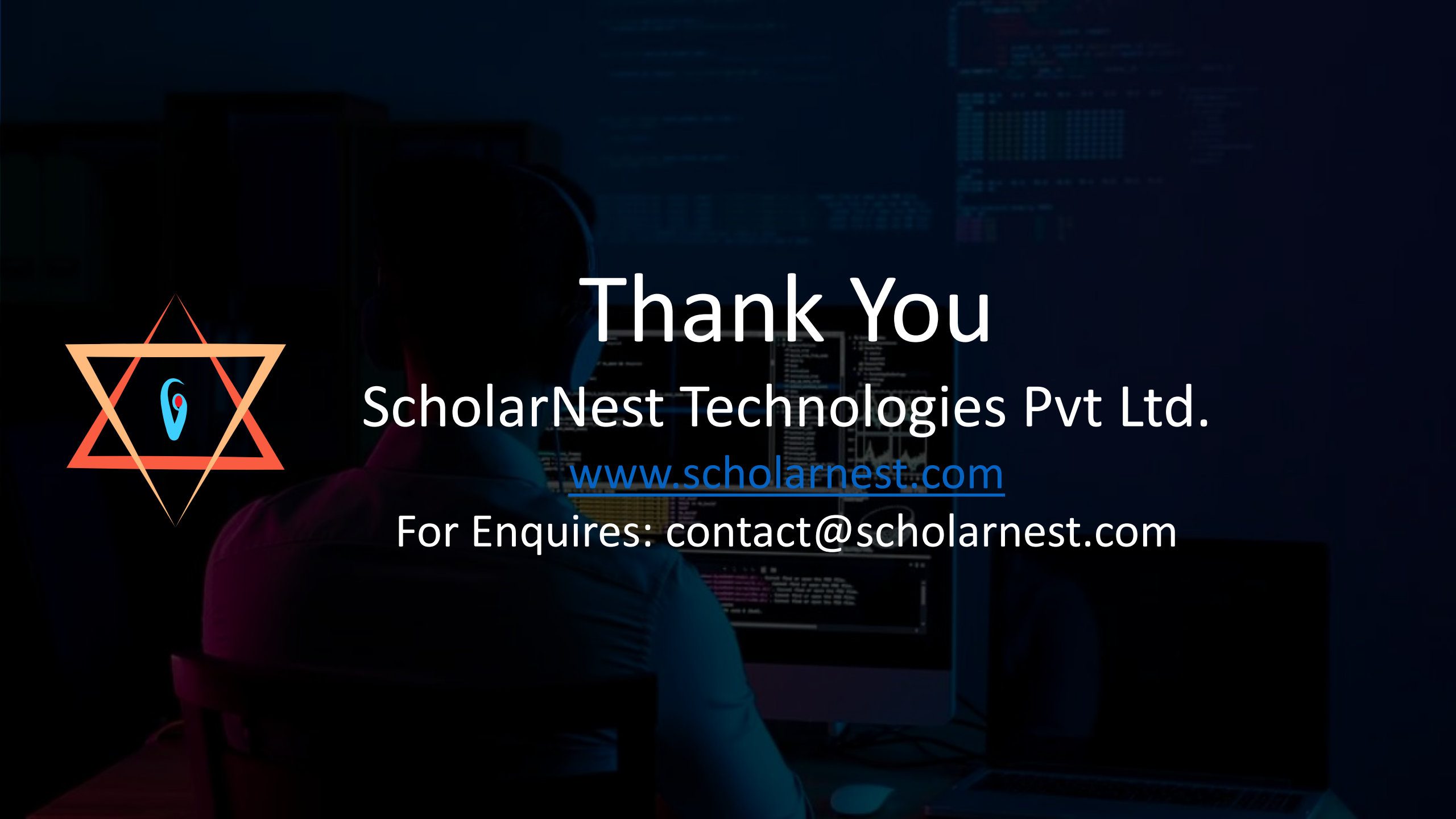


So if you look from the YARN perspective. The Spark container memory allocation looks like this. You have a container, and the container has got some memory. This total memory is broken into two parts:

1. **Heap memory (driver/executor memory)** – The heap memory goes to your JVM. We call it driver memory when you are running a driver in this container. Similarly, we call it executor memory when the container is running an executor.
2. **Overhead memory (OS Memory)** - The overhead memory is used for a bunch of things. In fact, the overhead is also used for network buffers. So you will be using overhead memory as your shuffle exchange or reading partition data from remote storage etc.



So, both the memory portions are critical for your Spark application.
And more than often, lack of enough overhead memory will cost you an OOM exception.
Because the overhead memory is often overlooked, but it is used as your shuffle exchange or network read buffer.



Thank You

ScholarNest Technologies Pvt Ltd.

www.scholarnest.com

For Enquires: contact@scholarnest.com

Spark Azure Databricks

Databricks Spark Certification and Beyond

Module:
Advanced Spark

Lecture:
Memory Management

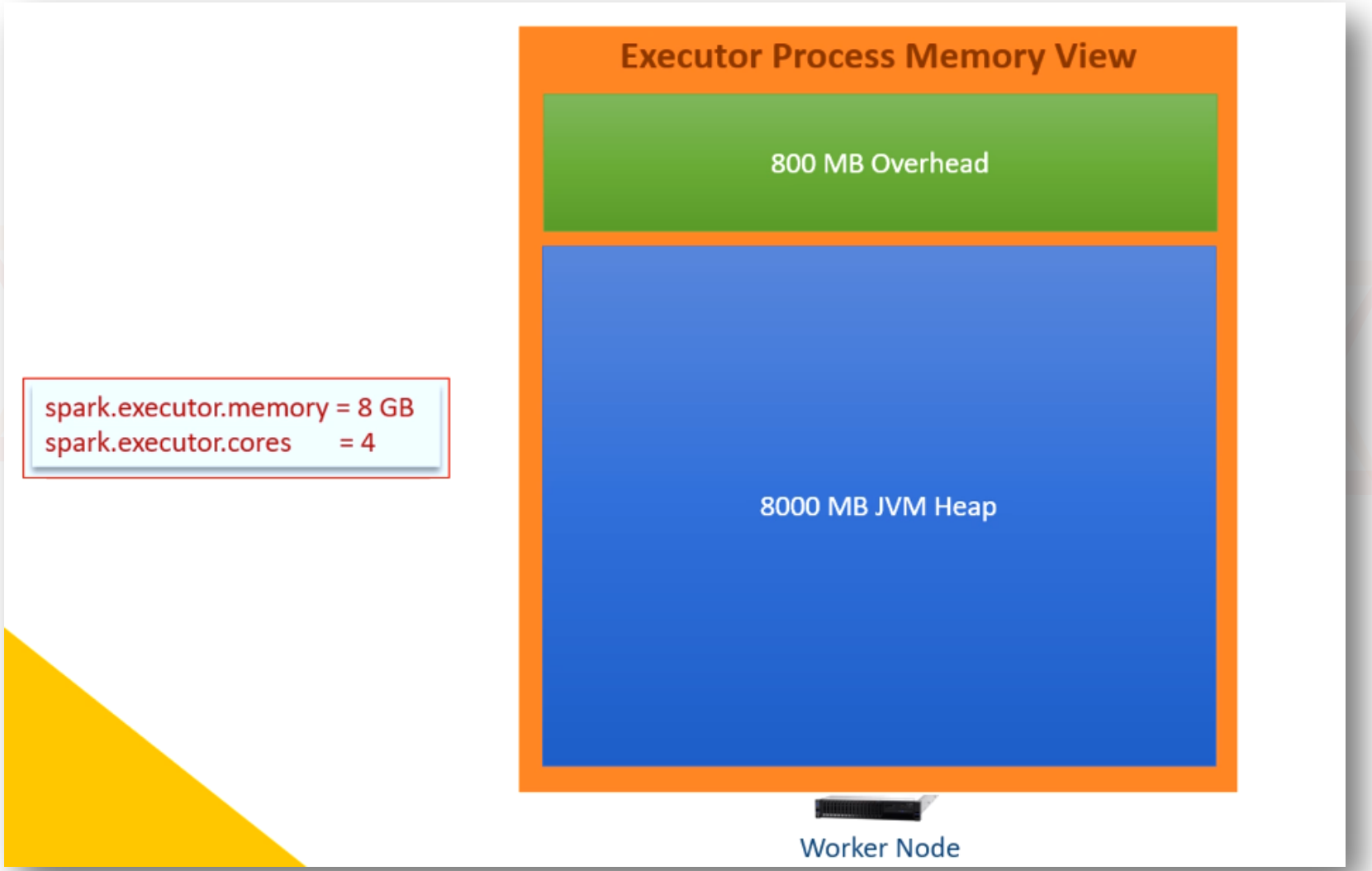
Spark Memory Management

In the previous video, I talked about container memory allocation.
We learned about the following:

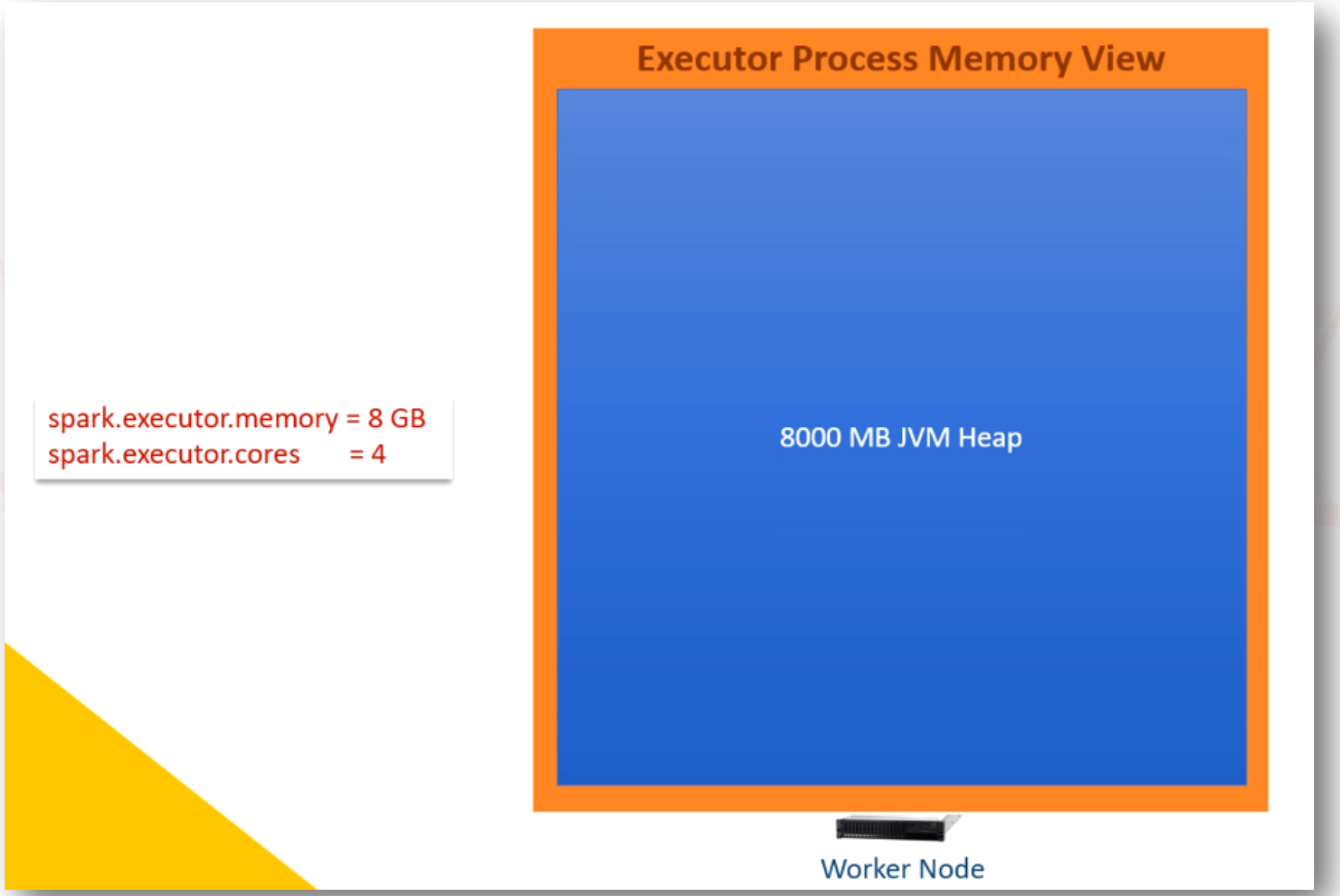
1. Overhead memory
2. Executor memory

In this video, we will learn more about executor memory.

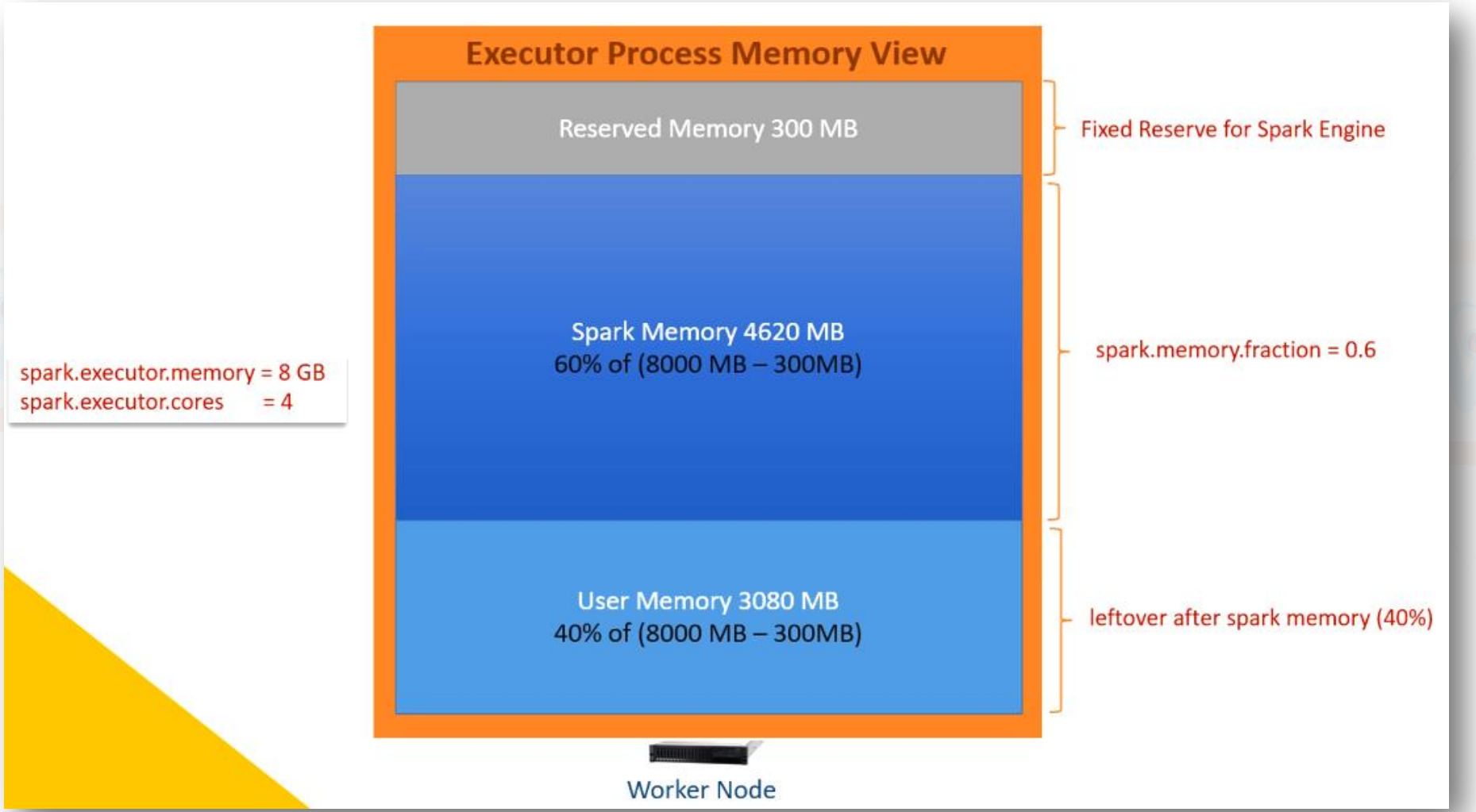
So let's start with the following assumption.
You started your application using *spark.executor.memory = 8 GB* and *spark.executor.cores = 4*
And you got 8 GB memory and 4 CPU cores. You will also get default 10% overhead memory.



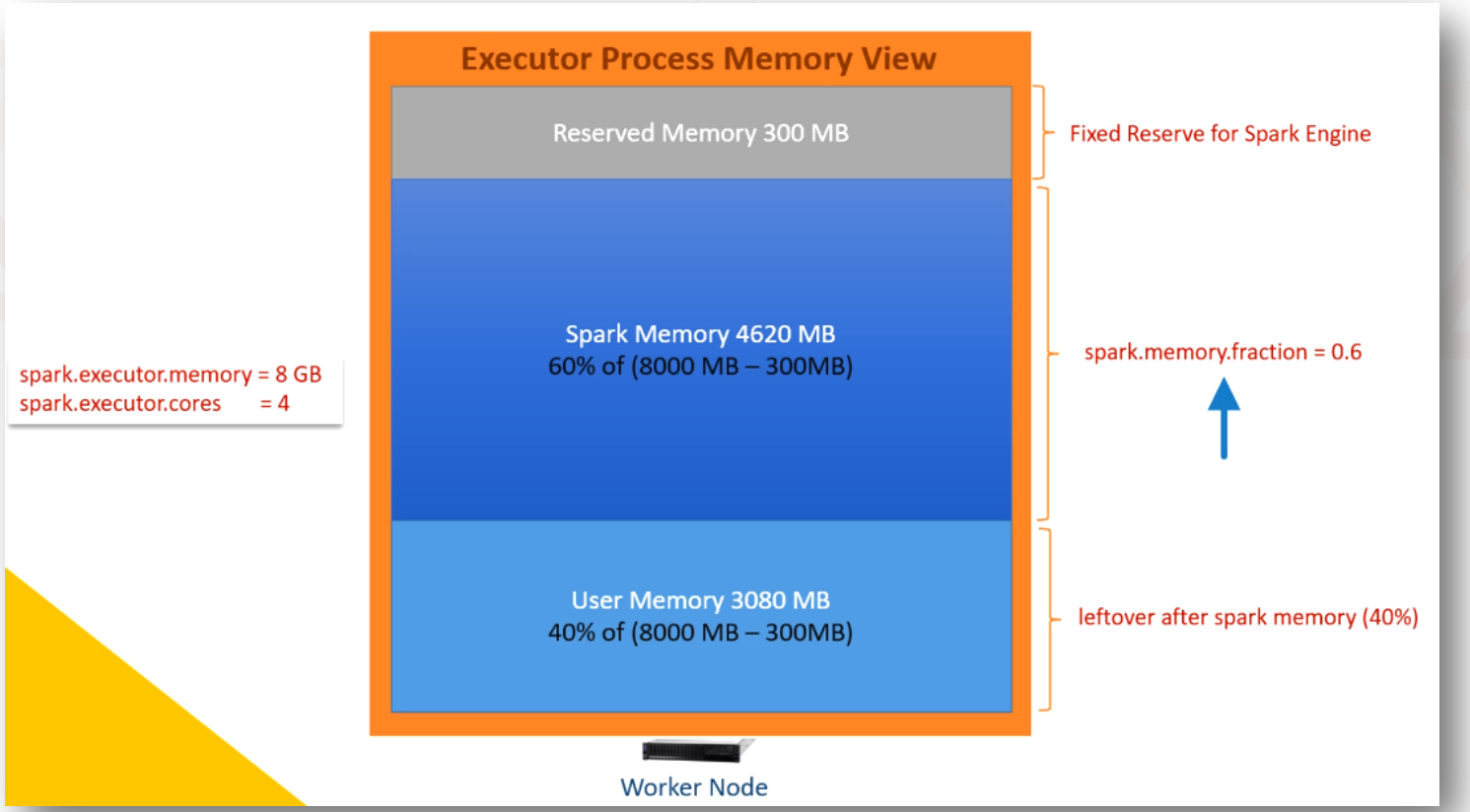
Let me remove this overhead memory from the equation and try to focus only on the JVM memory. We want to learn how Spark utilizes JVM heap memory.



The heap memory is further broken down into three parts as shown below.
So let's assume I got 8 GB for the JVM heap. This 8 GB is divided into three parts. Spark will reserve 300 MB for itself. That's fixed, and the Spark engine itself uses it.



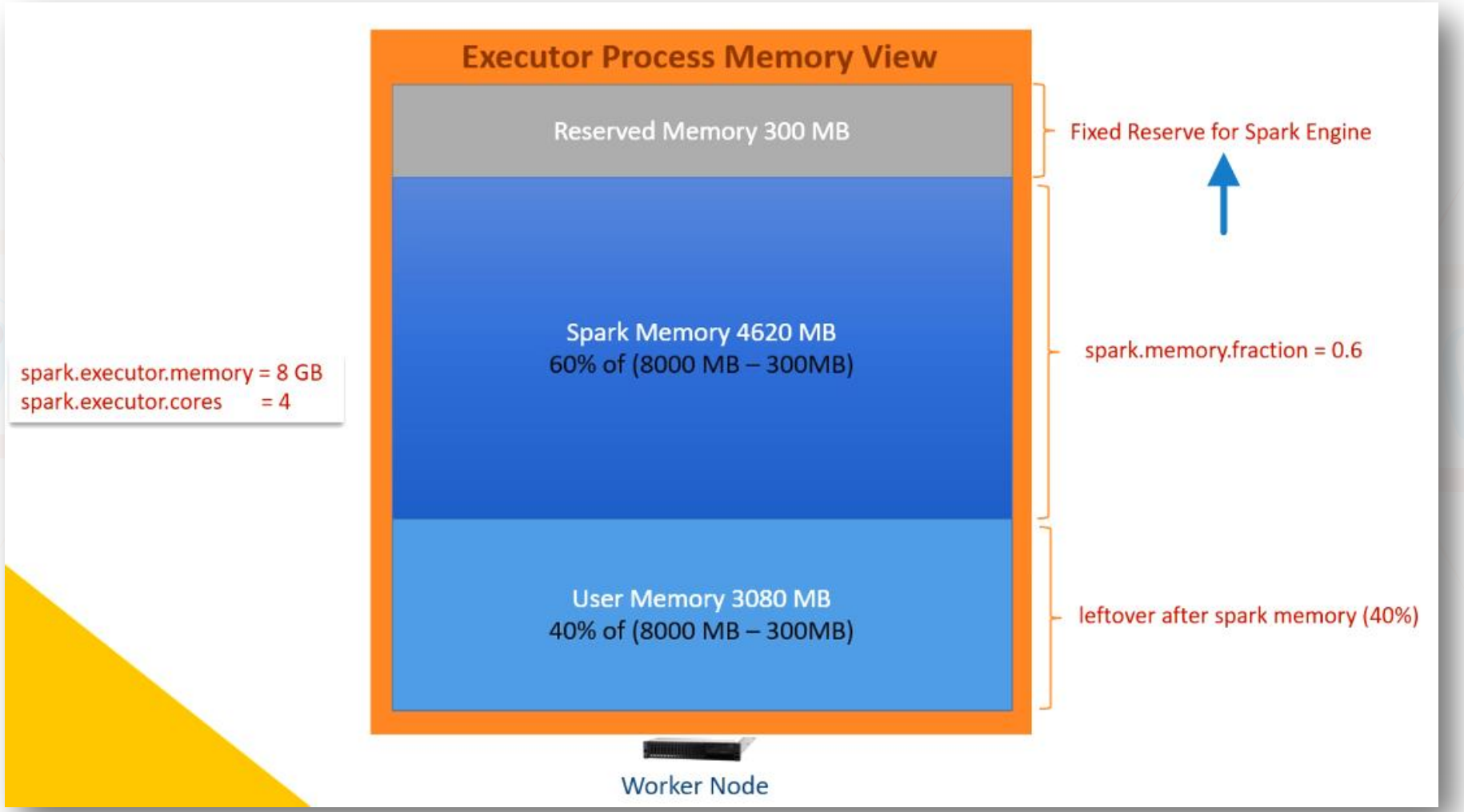
The next part is the Spark executor memory pool. This one is controlled by the *spark.memory.fraction* configuration, and the default value is 60%. So for our example, the spark memory pool translates to 4620 MB. How? We got 8 GB or 8000 MB. Three hundred is gone for Reserved memory. We are left with 7700 MB. Now you take 60% of this, and you will get 4620 MB for the Spark memory pool.
We still have 3080 MB left, and that one is gone for user memory.



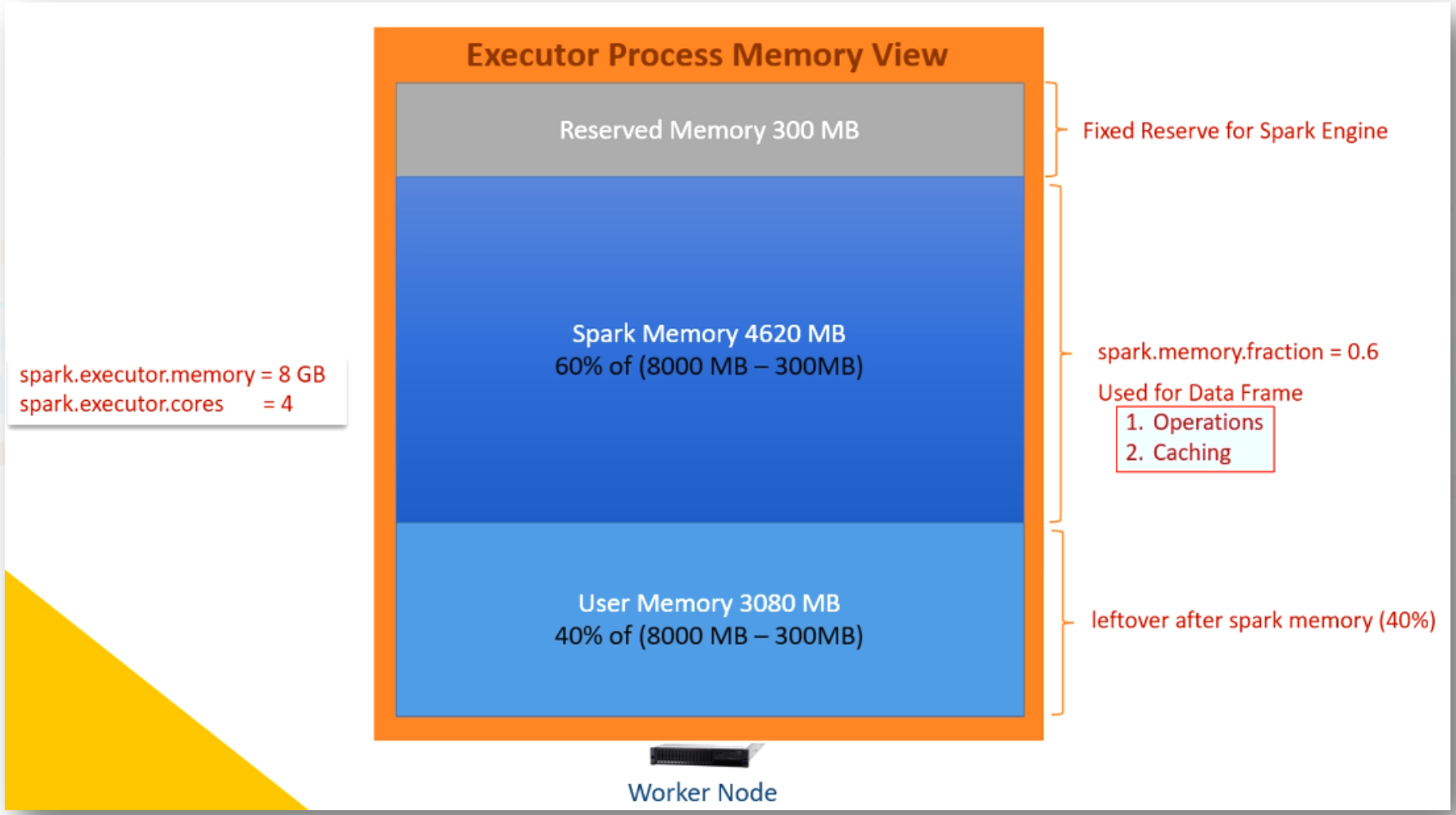
I hope you got the calculation for the memory division.
You can ask for executor memory using *spark.executor.memory* configuration.
We asked for 8 GB and got the same for the JVM heap.
Spark will reserve 300 MB, and you are left with 7700 MB.
This remaining memory is again divided into a 60-40 ratio.
The 60% goes to the Spark memory pool, and the remaining goes to the user memory pool.
If you want, you can change this 60-40 ratio using the *spark.memory.fraction* configuration.

Now let's try to understand these three memory pools.

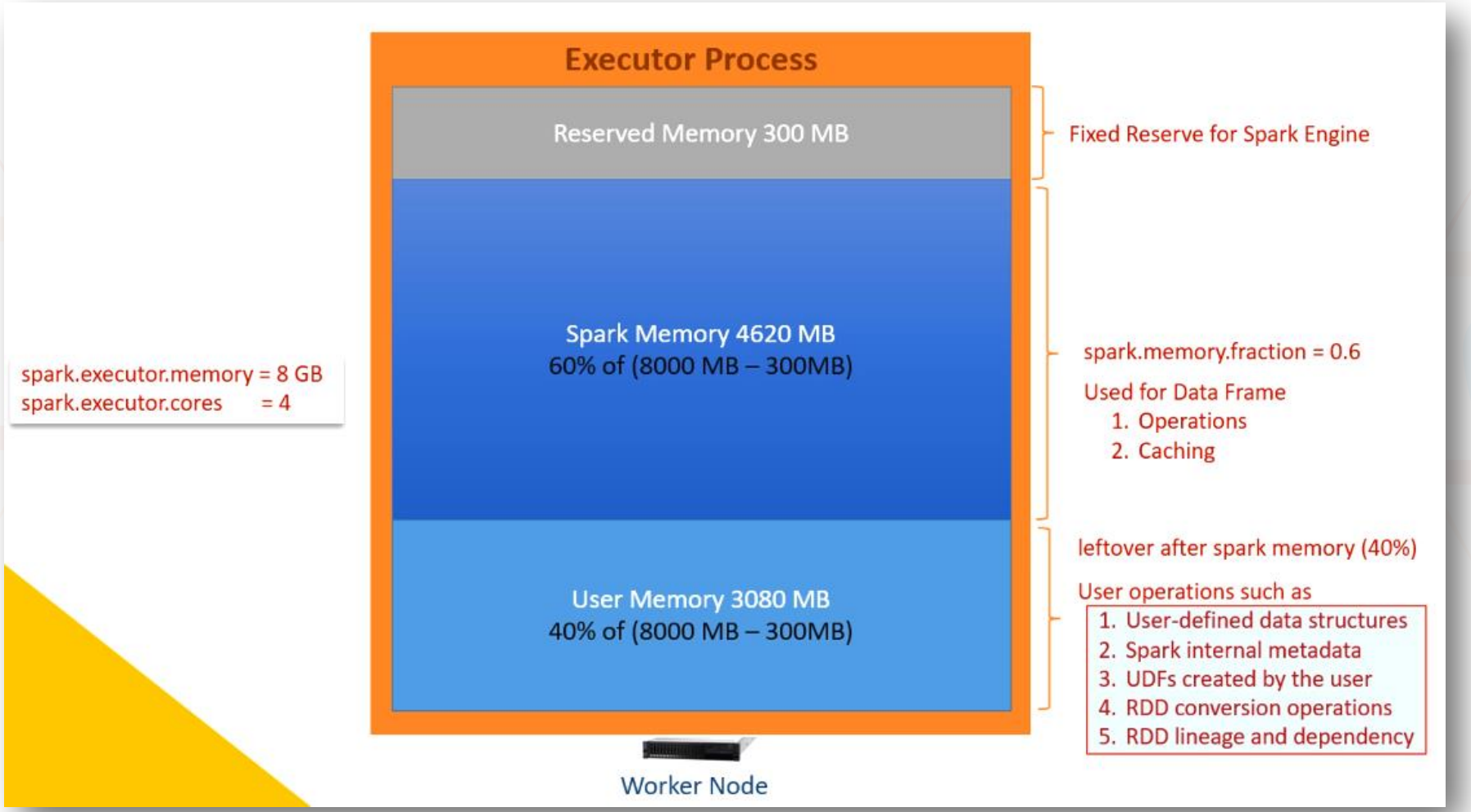
1. **Reserved Memory:** The reserved pool is gone for the Spark engine itself. You cannot use it.



2. **Spark Memory:** The Spark Memory pool is your main executor memory pool which you will use for data frame operations and caching.



3. **User Memory:** The User Memory pool is used for non-dataframe operations. Here are some examples for user operations highlighted below.

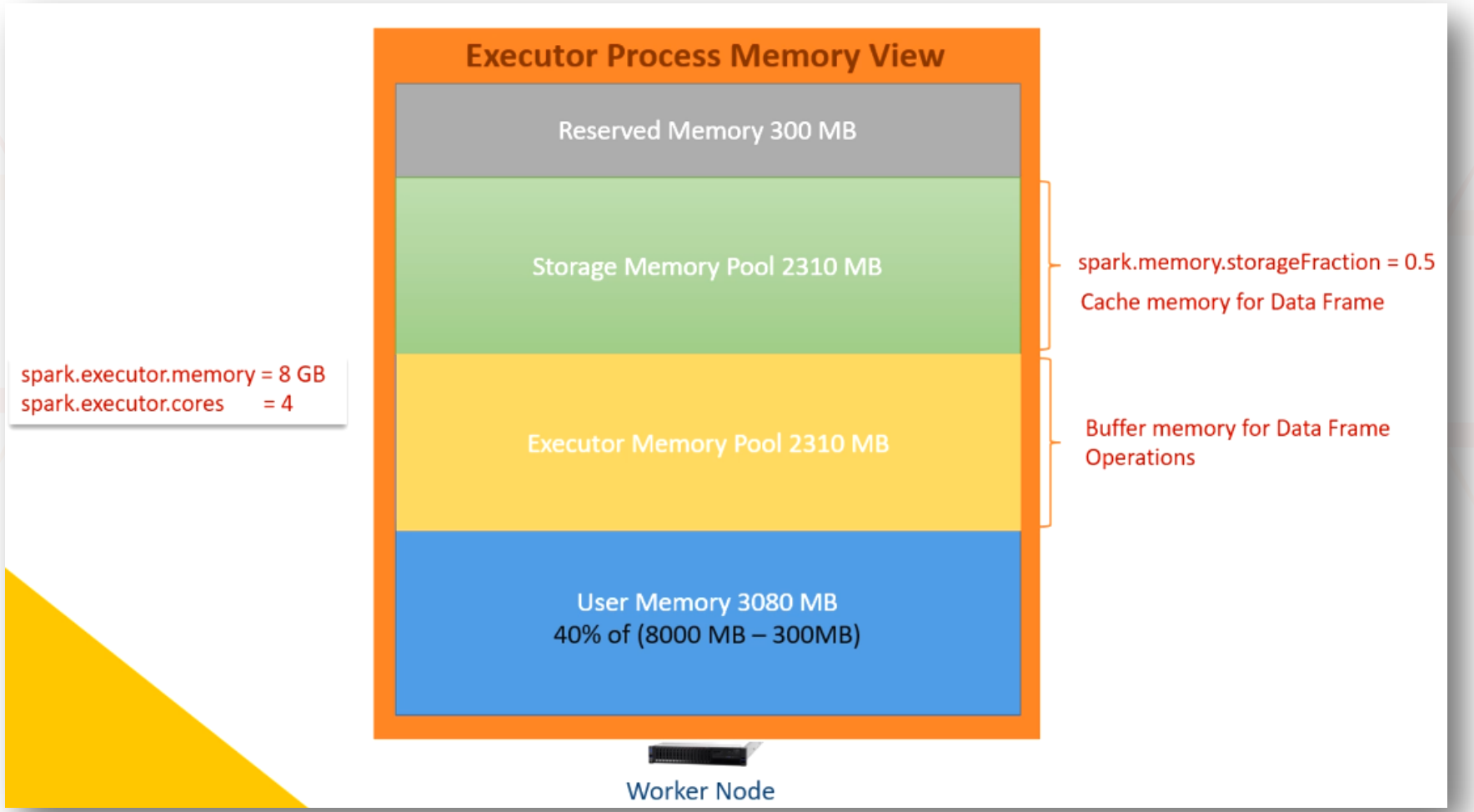


Examples for user operations:

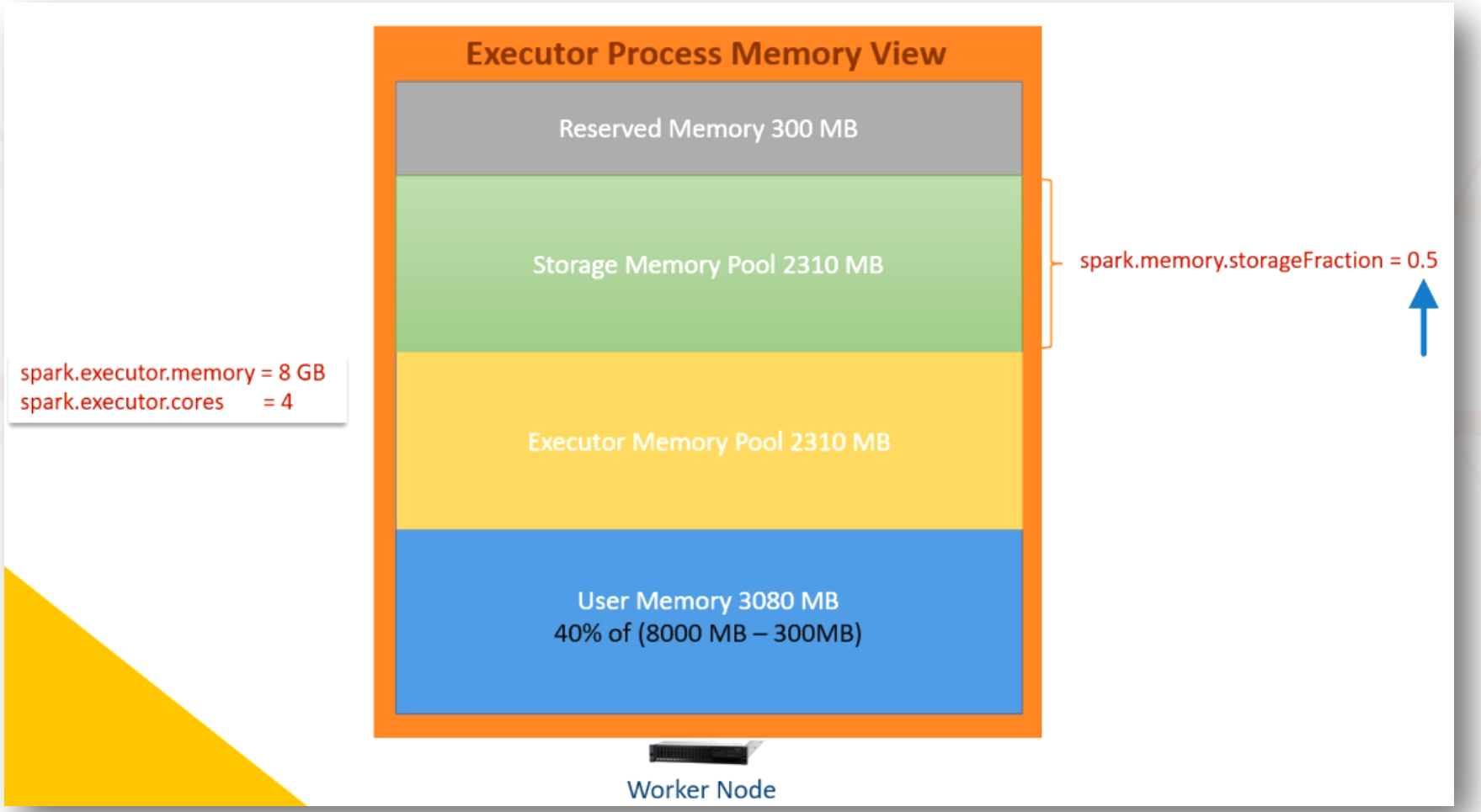
1. If you created user-defined data structures such as hash maps, Spark would use the user memory pool.
2. Similarly, spark internal metadata and user-defined functions are stored in the user memory.
3. All the RDD information and the RDD operations are performed in user memory.
4. But if you are using Dataframe operations, they do not use the user memory even if the Dataframe is internally translated and compiled into RDD.
5. You will be using user memory only if you apply RDD operations directly in your code.

The Spark memory pool is where all your data frames and Dataframe operations live.
You can increase it from 60% to 70% or even more if you are not using UDFs, custom data structures, and RDD operations. But you cannot make it zero or reduce it too much because you will need it for metadata and other internal things.

Now let's concentrate on the spark executor memory pool.
So for this example, we started with 8 GB, but we are left with a 4620 MB Spark memory pool.
This memory pool is further broken down into two sub-pools: Storage memory and Executor memory



The default break up is 50% each, but you can change it using the *spark.memory.storageFraction* configuration. So Spark will reserve 50% of the memory pool for the storage pool, and the remaining 50% comes to the executor pool.
For our example, we got a 2310 MB storage pool and another 2310 MB executor pool.



Now the next question is this:

What do we do in these two memory pools?

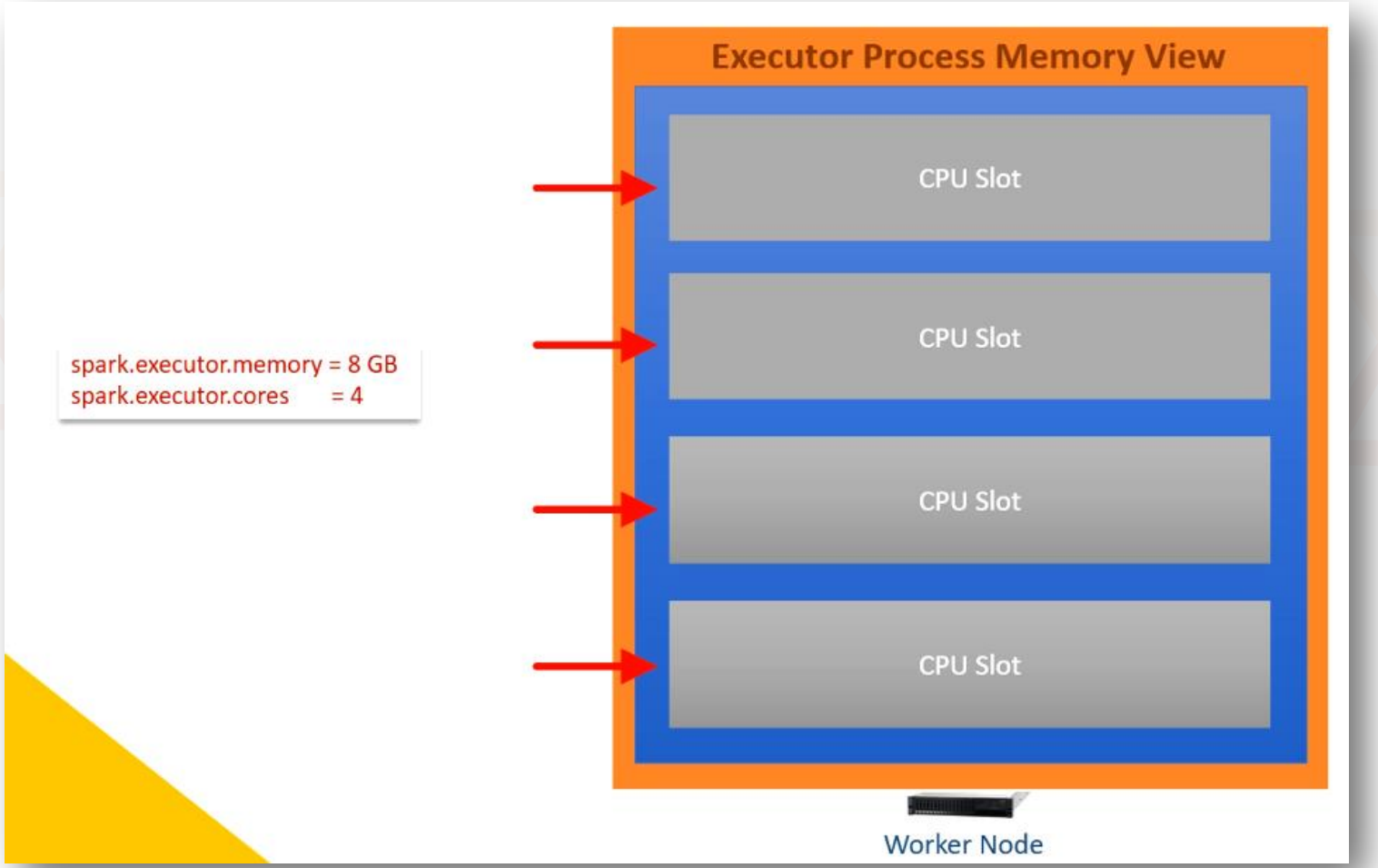
We use the storage pool for caching data frames. And the Executor pool is to perform Dataframe computations. So if you are joining two data frames, Spark will need to buffer some data for performing the joins. That buffer is created in the executor pool.

Similarly, if you are aggregating or performing some calculation, the required buffer memory comes from the executor pool. So executor pool is short-lived. You will use it for execution and free it immediately as soon as your execution is complete.

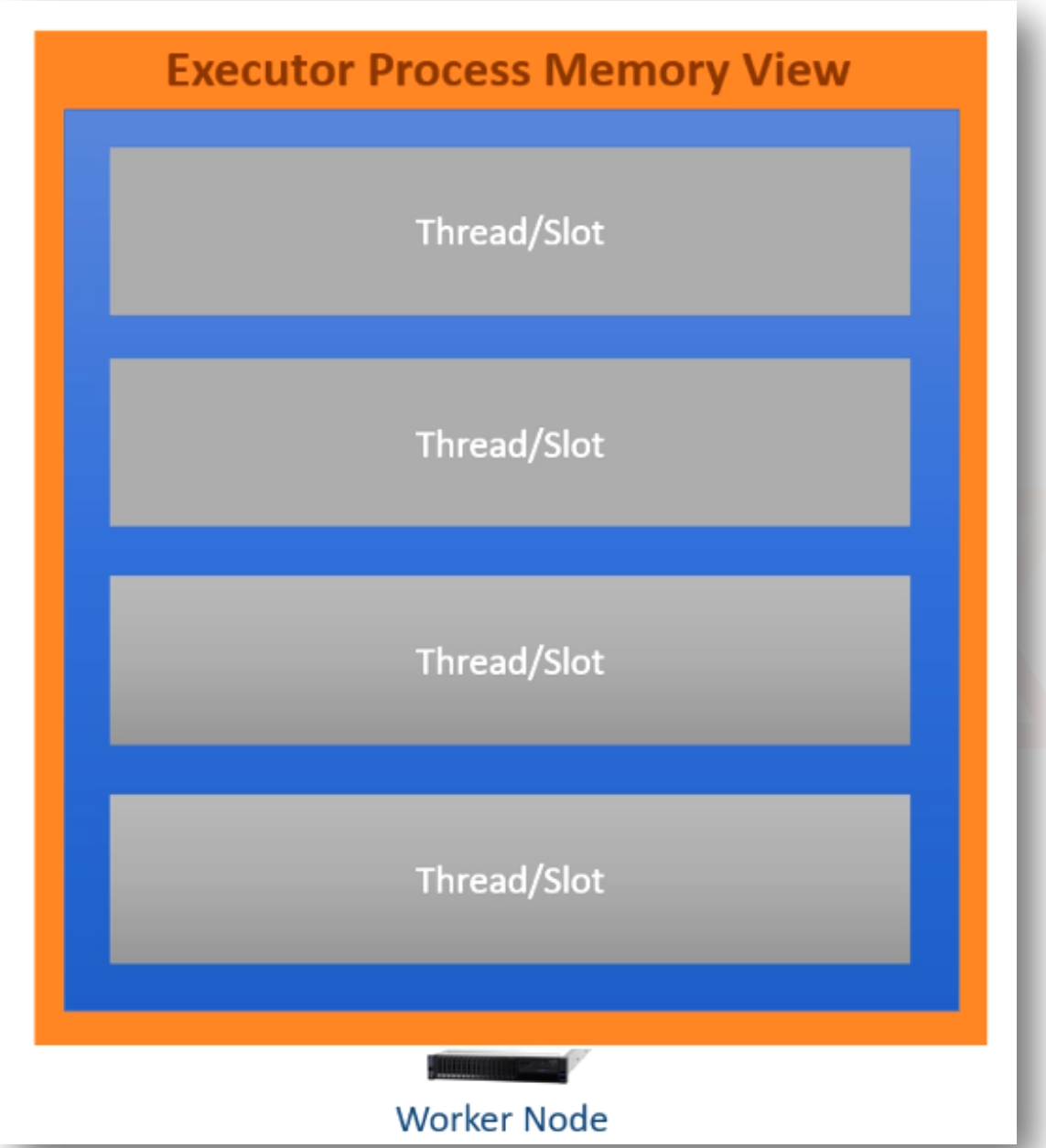
The storage pool is used to cache the data frames. So if you are using Dataframe cache operation, you will be caching the data in the storage pool.

So the storage pool is long-term. You will cache the Dataframe and keep it there as long as the executor is running or you want to un-cache it.

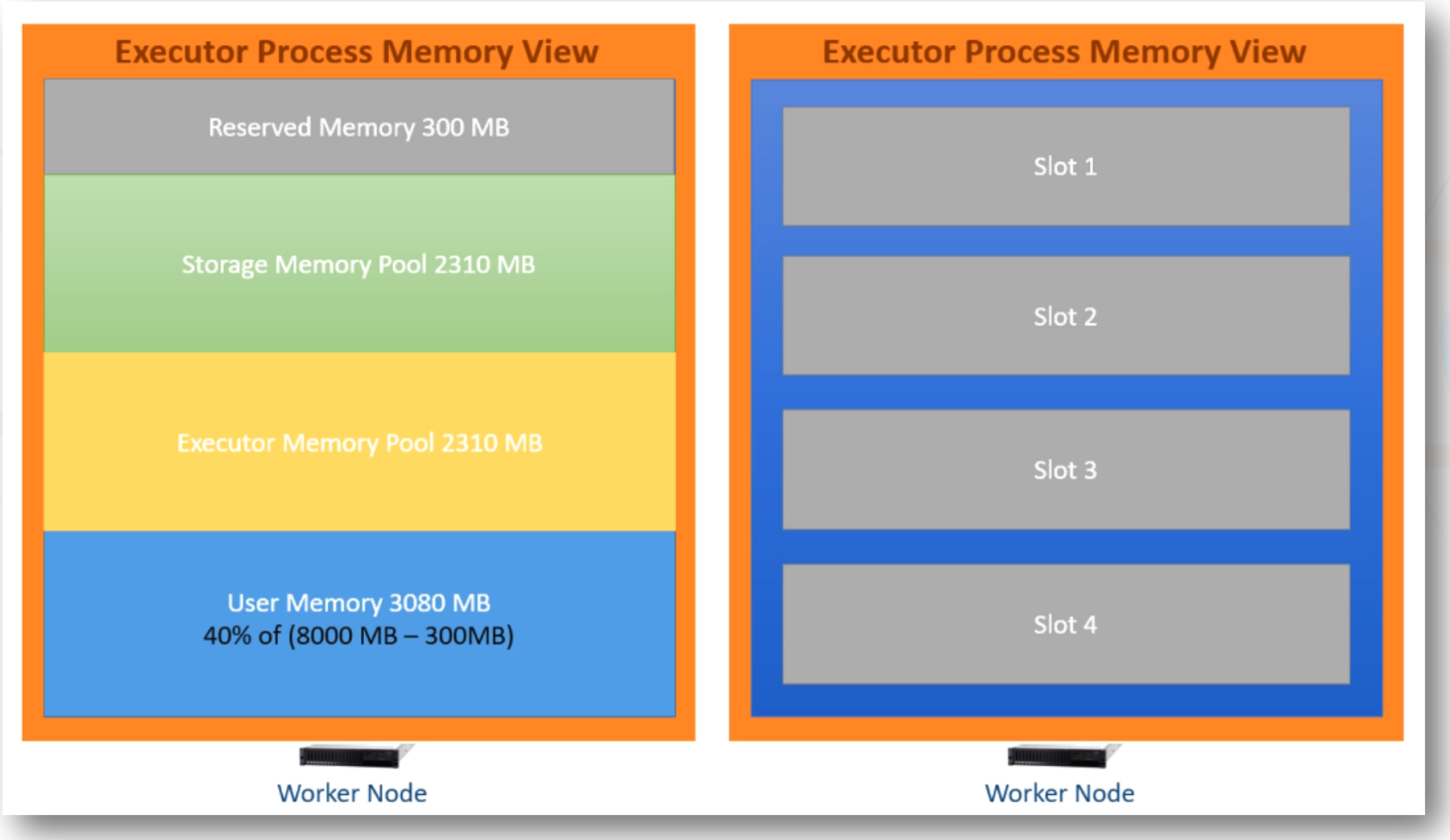
So we are seeing the memory view of the Spark executor. But we also have an angle of the CPU Cores. So let me bring that also into the discussion and talk about the compute-view of the executor. For this example, I asked for the 4 CPU cores. So my executor will have four slots, and we can run four parallel tasks in these slots.



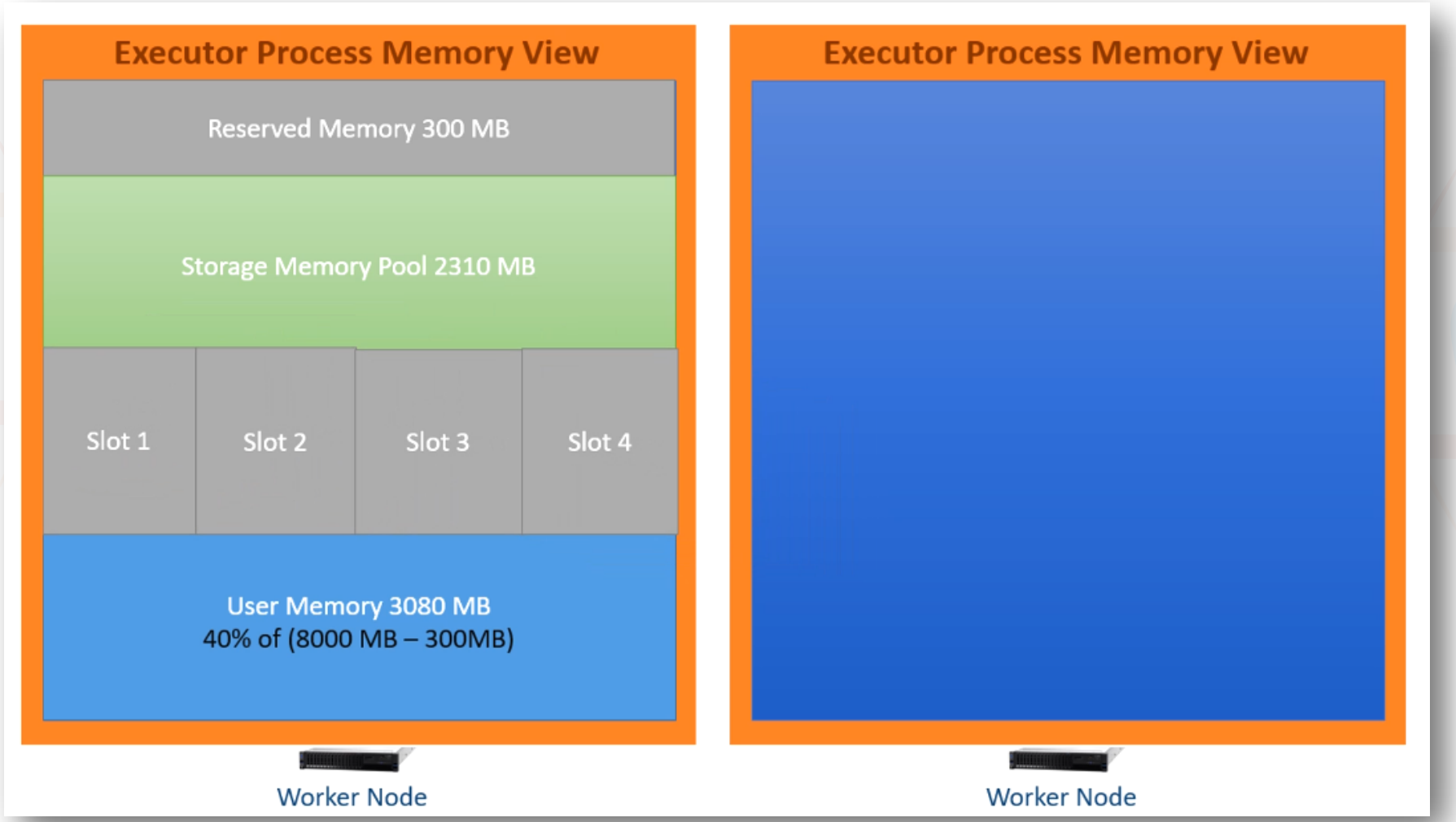
But what are these slots?
These slots are threads.
All these slots are threads within the same JVM.
We do not have multiple processes or JVMs here.
We have a single JVM, and all the slots are simply threads in the same JVM.



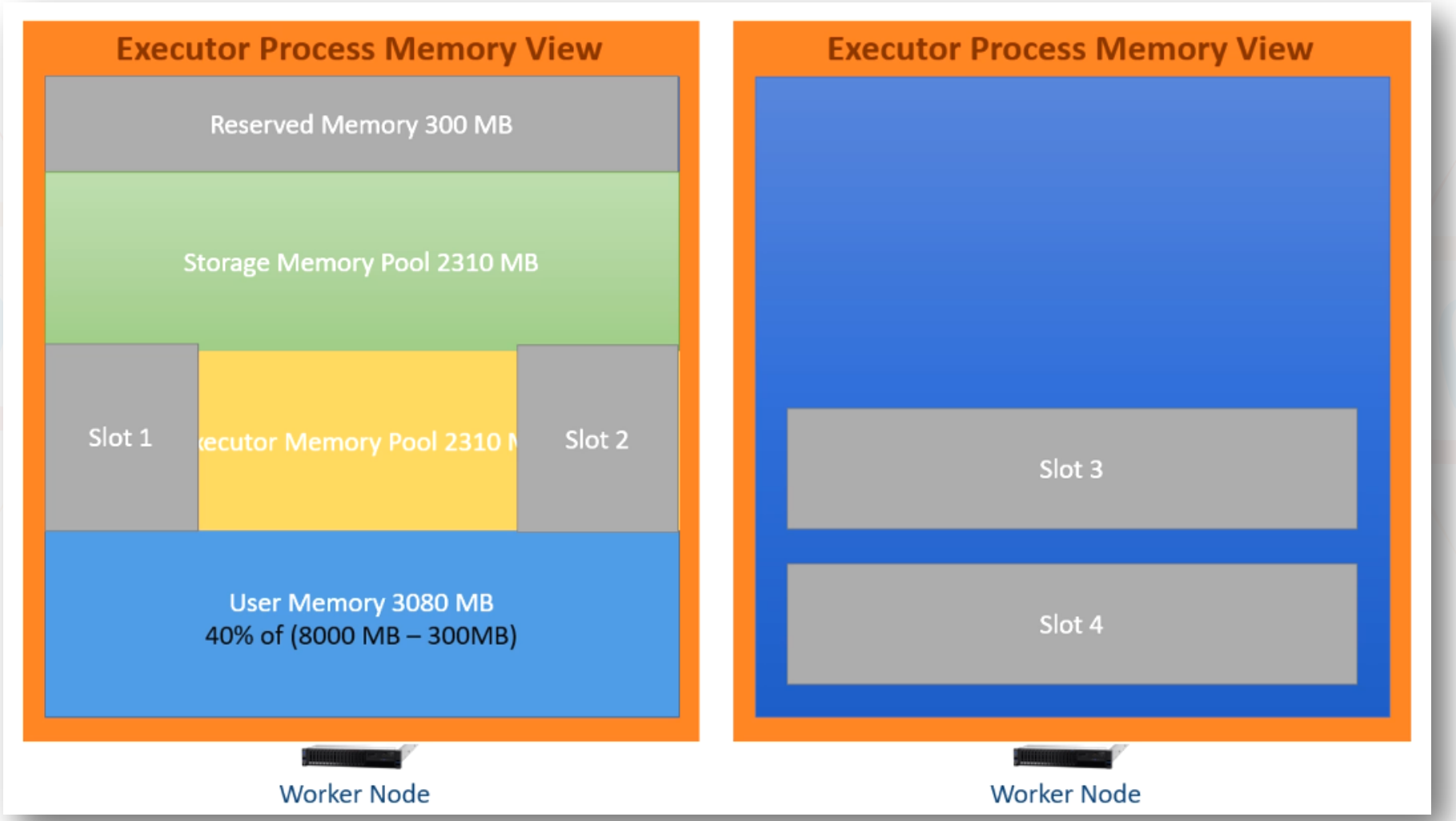
So I have one executor JVM, 2310 MB storage pool, another 2310 MB executor pool, and four threads to share these two memory pools.
So what do you think, how much executor memory will each task get? Simple! 2310/4.



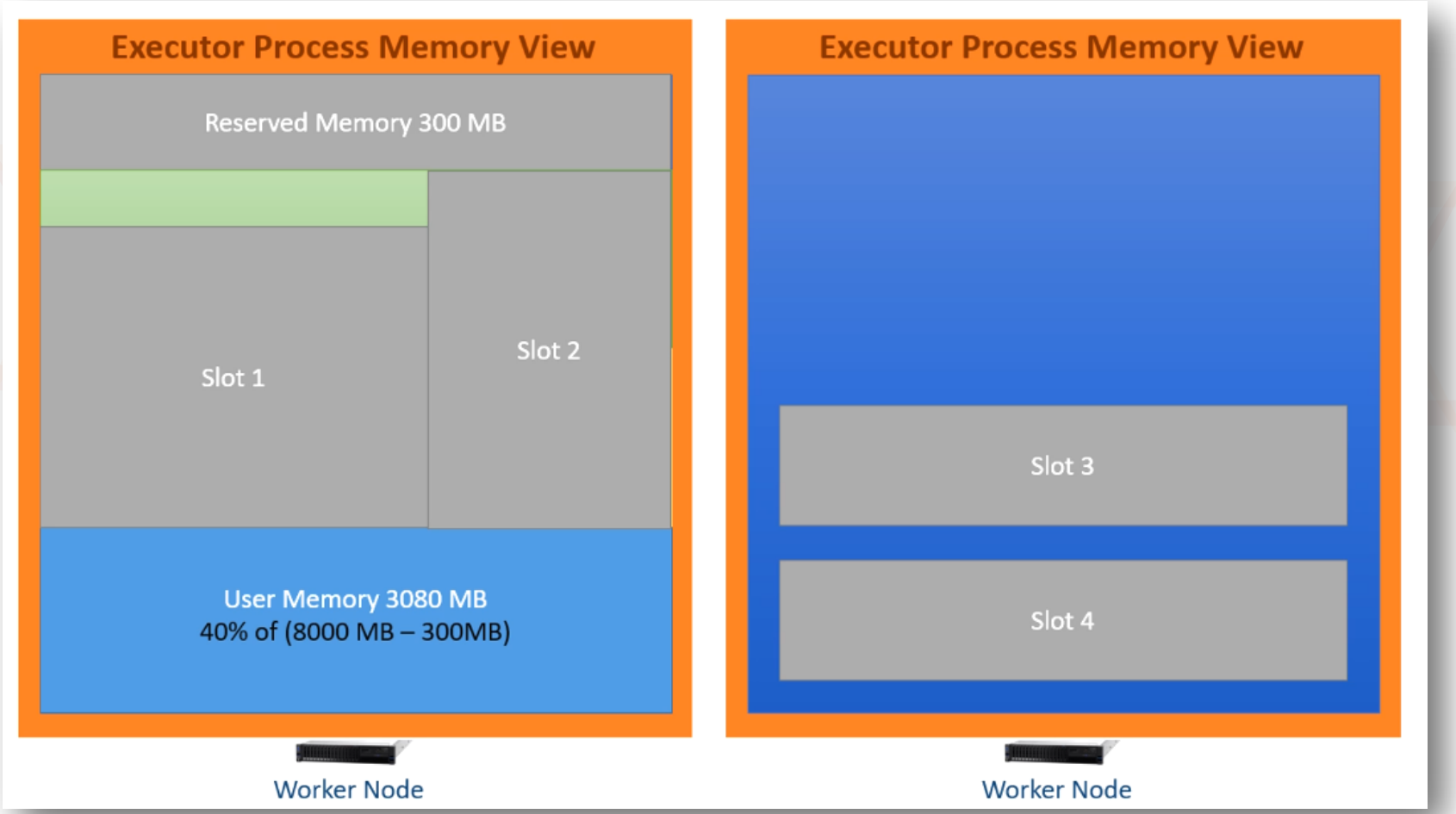
That's static memory management, and Spark used to assign task memory using this static method before spark 1.6. But now, they changed it and implemented a unified memory manager. The unified memory manager tries to implement fair allocation amongst the active tasks.



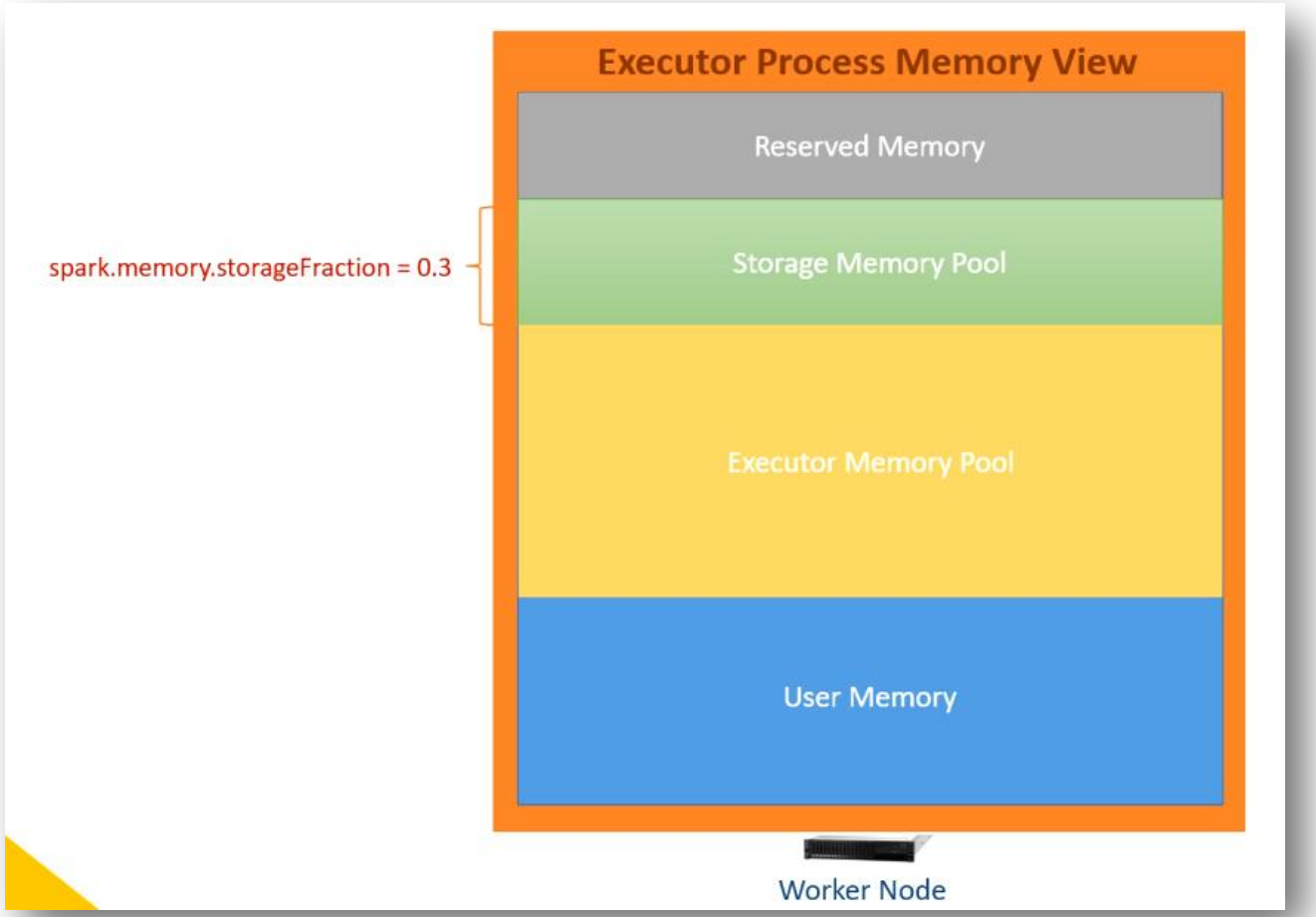
Let's assume I have only two active tasks. I have four slots, but I have only two active tasks. So the unified memory manager can allocate all the available execution memory amongst the two active tasks.



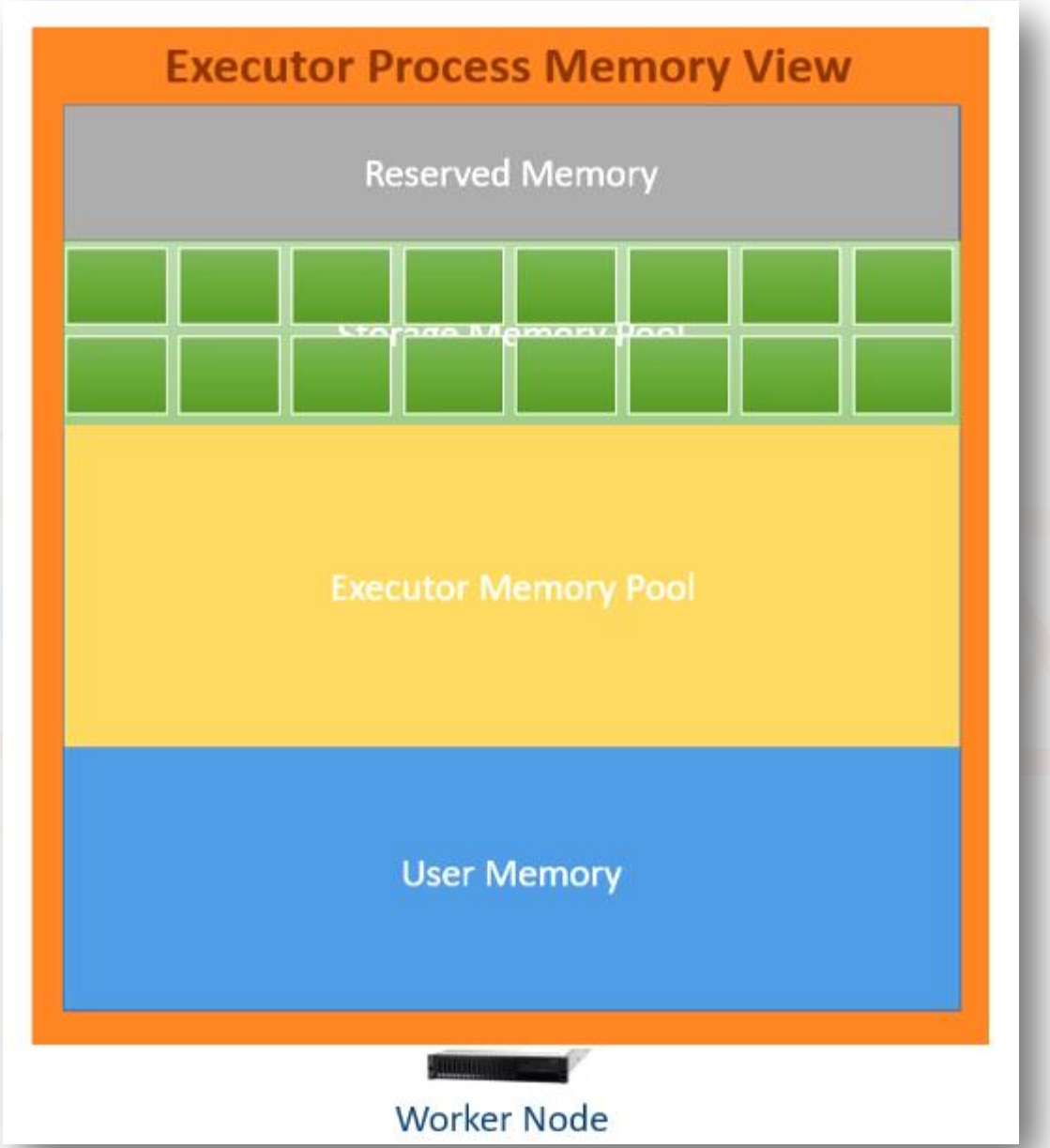
So, there is nothing reserved for any task. The task will demand the execution memory, and the unified memory manager will allocate it from the pool. If the executor memory pool is fully consumed, the memory manager can also allocate executor memory from the storage memory pool as long as we have some free space. What does it mean?



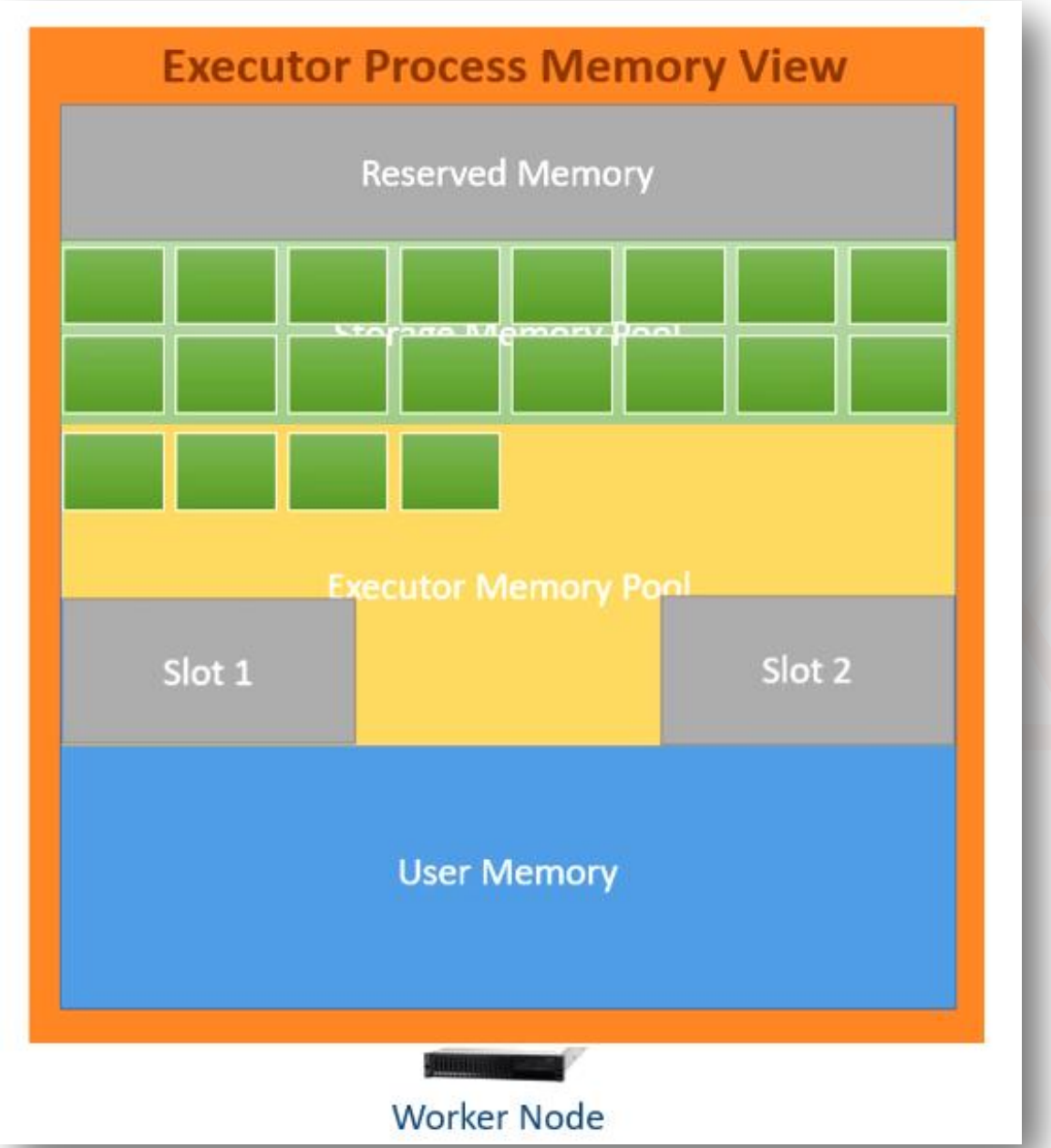
It means, we start with a 50-50 boundary between executor memory and storage memory. But you can change it using *spark.memory.storageFraction*. I know I will not cache too many data frames. So I can reduce it to 30%. So I shifted the boundary. But this boundary is flexible, and we can borrow space as long as we have free space. So the memory manager can borrow some memory from the other side as long as the other side is free.



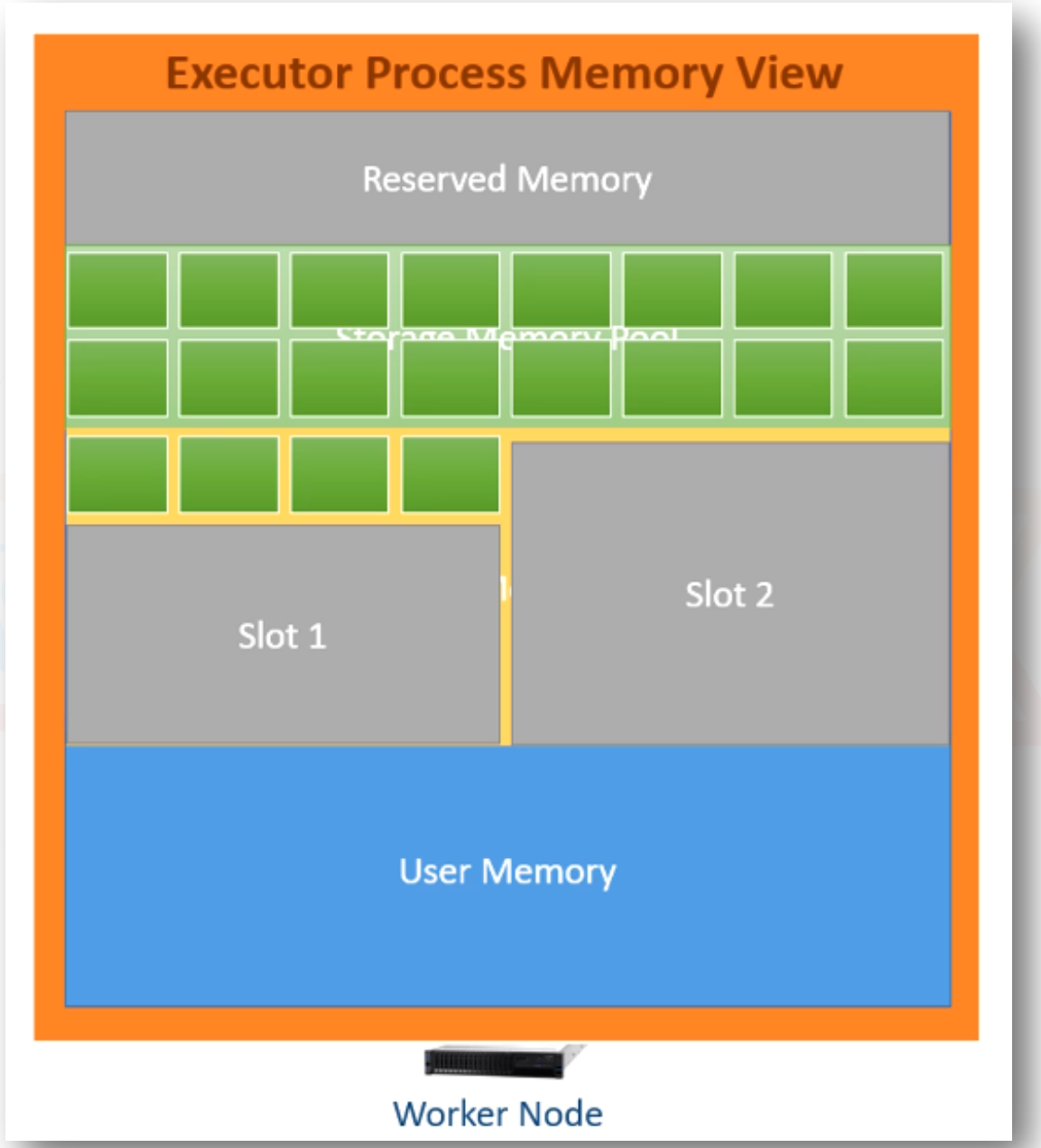
Let's assume we cached some data frames and entirely consumed the storage memory pool.
But I want to cache some more data.
So the memory manager will give me some more memory from the executor memory pool.



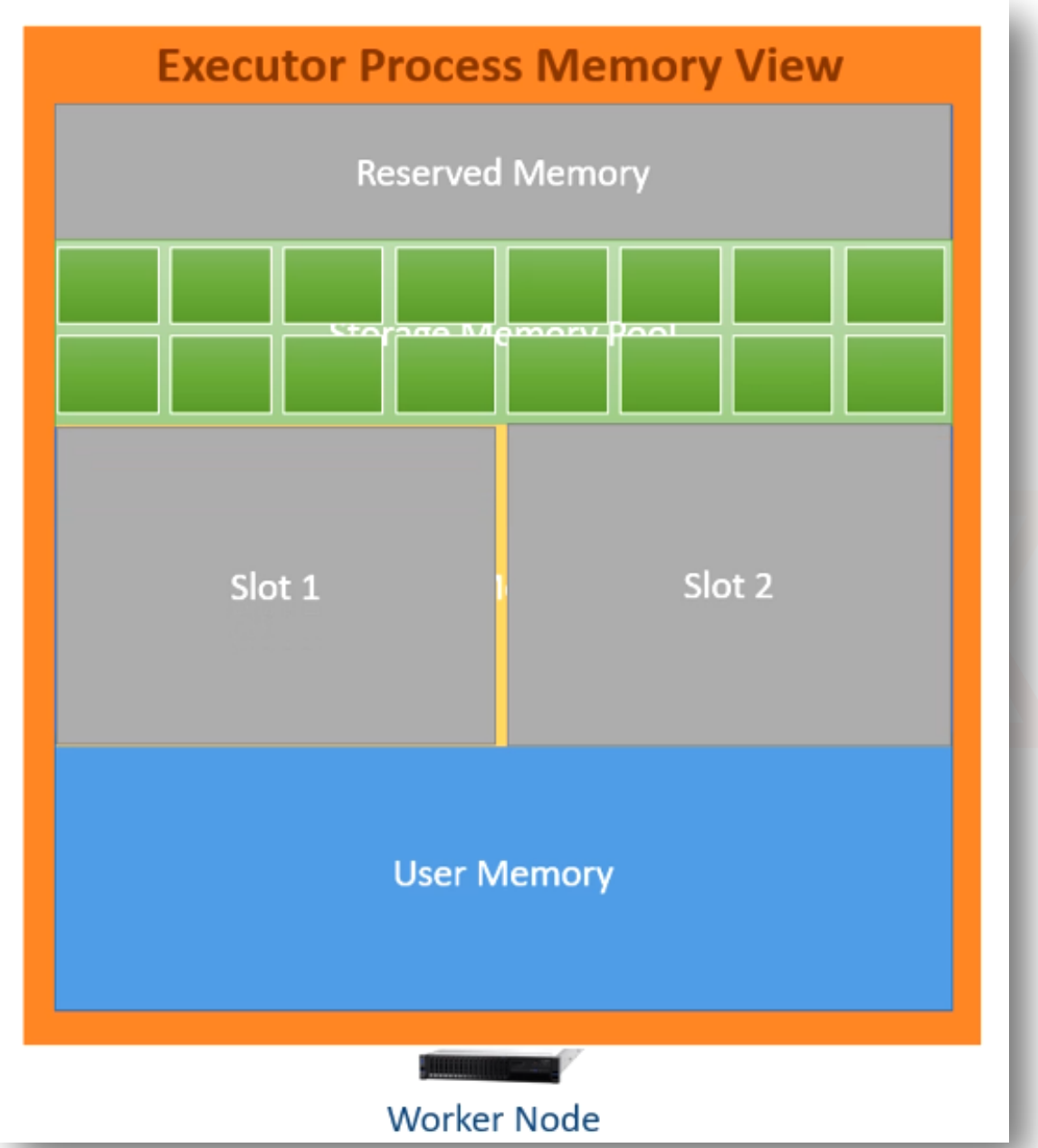
The executor memory was free, so we consumed it.
Now executor is performing some join operation, and it needs memory.



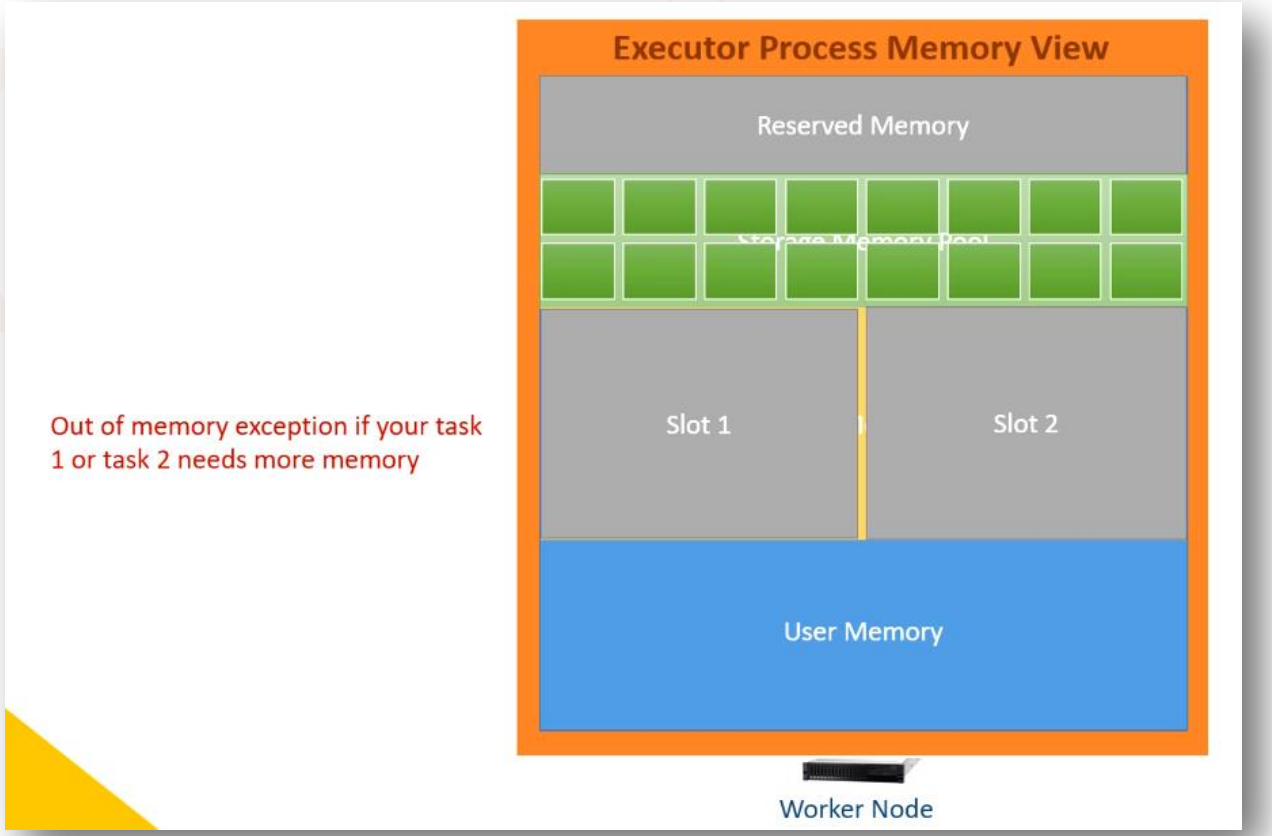
So the executor will start consuming memory as long as there is free space.



But in the end, the memory manager will evict the cached data frames and spill them to the disk to make more space for the executor.
We borrowed it from the executor pool. Right? And now the executor needs it. So the memory manager will evict it.



However, you reached the boundary, and the executor needs more memory. This boundary is rigid, and the memory manager cannot evict any changed data blocks. If it was free, the memory manager should have given it to the executor. But it is not free, and we already occupied it. However, the executor needs more memory. How do you manage it? The executor will try to spill few things to the disk and make some more room for itself. What if you have things that you cannot spill to disk? Congratulations! You are hitting the OOM exception.



The Spark memory management is too complex. But you should understand the structure so you can manage things more efficiently. And you have the following configurations to control it.

1. *spark.executor.memoryOverhead* – It will give you 10% extra of whatever you are asking for the JVM, and it is outside the JVM and reserved for overhead.
2. *spark.executor.memory* – Your JVM memory comes from Spark.executor.memory. This configuration allows you to ask for the executor JVM memory. Whatever you get, the spark engine will reserve 300 MB from your allocation.
3. *spark.memory.fraction* – Now you can configure spark.memory.fraction to tell how much you want to use for Dataframe operations. The default value is 60%, but you can increase it. The leftover is kept aside for non-dataframe operations, and we call it user memory.

4. *spark.memory.storageFraction* – The spark.memory.storageFraction allows you to set a hard boundary that the memory manager cannot evict. If you are not dependent on cached data, you can reduce this hard limit.
5. *spark.executor.cores* – Finally, you can ask for the executor cores. The executor core is critical because this configuration defines the maximum number of concurrent threads. If you have too many threads, you will have too many tasks competing for memory. If you have a single thread, you might not be able to use the memory efficiently. In general, Spark recommends two or more cores per executor, but you should not go beyond five cores. More than five cores cause excessive memory management overhead and contention, so they recommend stopping at five cores.

So we learned about the JVM heap. Spark also allows you the following configurations:

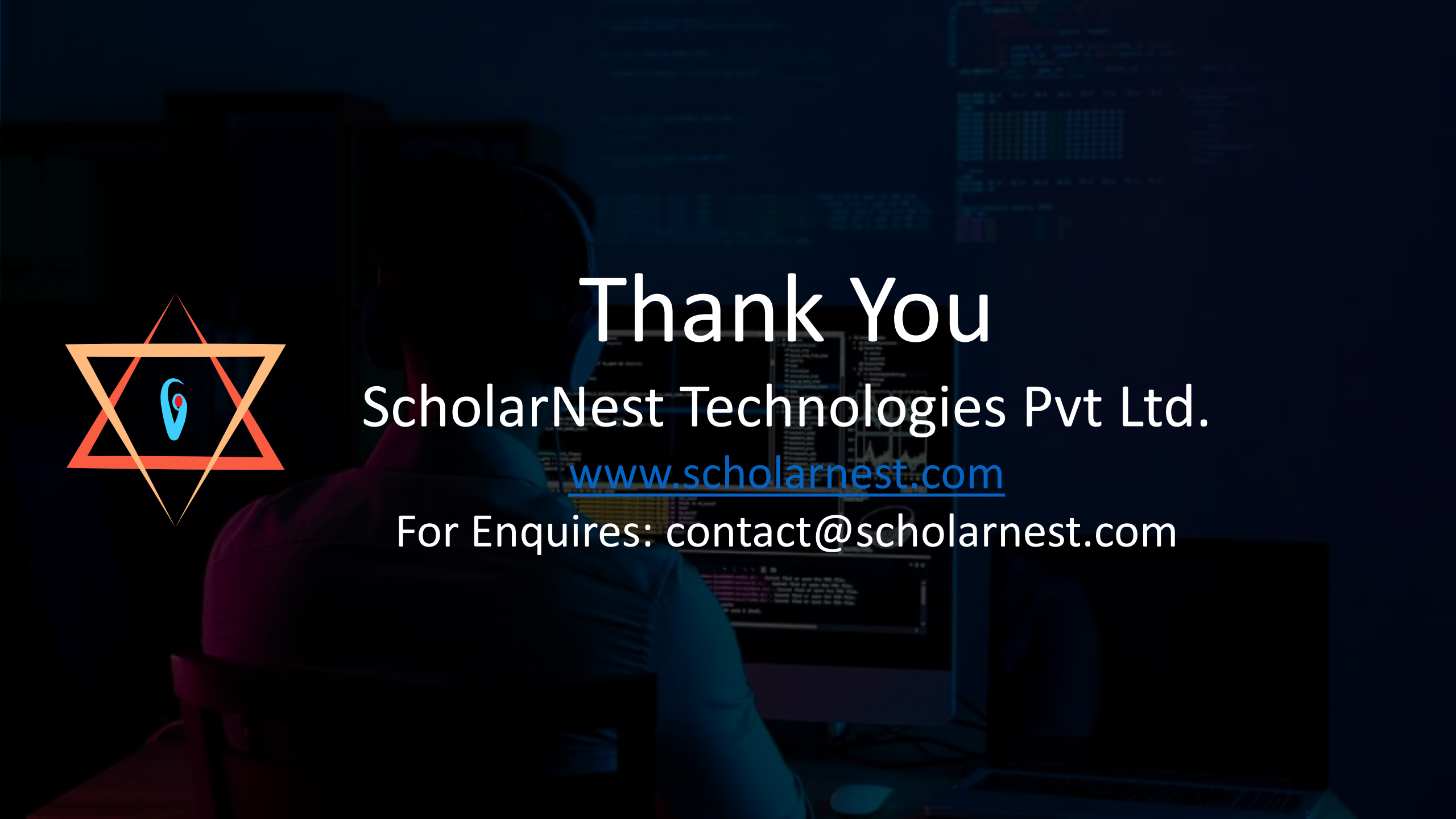
1. spark.executor.pyspark.memory
2. spark.memory.offHeap.enabled
3. spark.memory.offHeap.size

These three configurations will give you off-heap memory outside your JVM.
What does it mean? Let's try to understand.
I talked about the overhead memory, and we also talked about the heap memory.
Now let's talk about the off-heap memory.

We already learned that most of the spark operations and data caching are performed in the JVM heap. And they perform best when using the on-heap memory.
However, the JVM heap is subject to garbage collection. So if you are allocating a huge amount of heap memory to your executor, you might see excessive garbage collection delays.
However, Spark 3.x was optimized to perform some operations in the off-heap memory.
Using off-heap memory gives you the flexibility of managing memory by yourself and avoid GC delays. Spark can take some advantage of this idea and use off-heap memory.

So if you need an excessive amount of memory for your Spark application, it might help you take some off-heap memory. For large memory requirements, mixing some on-heap and off-heap might help you reduce GC delays. By default, the off-heap memory feature is disabled. You can enable it by setting *spark.memory.offHeap.enabled = true*.
And you can set your off-heap memory requirement using *spark.memory.offHeap.size*.
So, where is this off-heap memory used? Spark will use the off-heap memory to extend the size of spark executor memory and storage memory. What does it mean? The off-heap memory is some extra space. So if needed, Spark will use it to buffer spark Dataframe operations and cache the data frames. So adding off-heap is an indirect method of increasing the executor and storage memory pools.

Now the last item is the *spark.executor.pyspark.memory.*
I hope you already learned that Apache Spark is written in Scala.
Scala is a JVM language, and hence Spark is also a JVM application.
But if you are using PySpark, your application may need a Python worker.
These Python workers cannot use JVM heap memory. So they use off-heap overhead memory.
But what if you need more off-heap overhead memory for your Python workers?
You can set that extra memory requirement for your Python workers using *spark.executor.pyspark.memory*.
We do not have a default value for this configuration because most of the PySpark applications do not use external Python libraries, and they do not need a Python worker.
So, Spark does not set a default value for this configuration parameter.
But if you need some extra memory for your Python workers, you can set the requirement using the *spark.executor.pyspark.memory.*



Thank You

ScholarNest Technologies Pvt Ltd.

www.scholarnest.com

For Enquires: contact@scholarnest.com

Spark Azure Databricks

Databricks Spark Certification and Beyond

Module:
Advanced Spark

Lecture:
Data Caching

Data Caching in Spark

In this chapter, I will talk about caching Spark data frames in the memory.
We already learned about the Spark memory pool in an earlier video.
Spark memory pool is broken down into two sub-pools:

1. Storage memory
2. Executor memory

We also learned the purpose of these two memory pools. Spark will use the Executor pool to perform Dataframe computations. And we can use the storage pool for caching data frames.

Answers to all these following questions are essential and critical for any Spark developer. And that's what we are going to explore and learn in this video.

1. How do you cache?
2. Why should you cache?
3. When cache and when not cache?
4. How to un-cache?
5. Caching formats?
6. Caching memory or disk?

So let's start with the following question:

How do we cache a Dataframe in the executor storage pool?

Well, we have two methods for doing this:

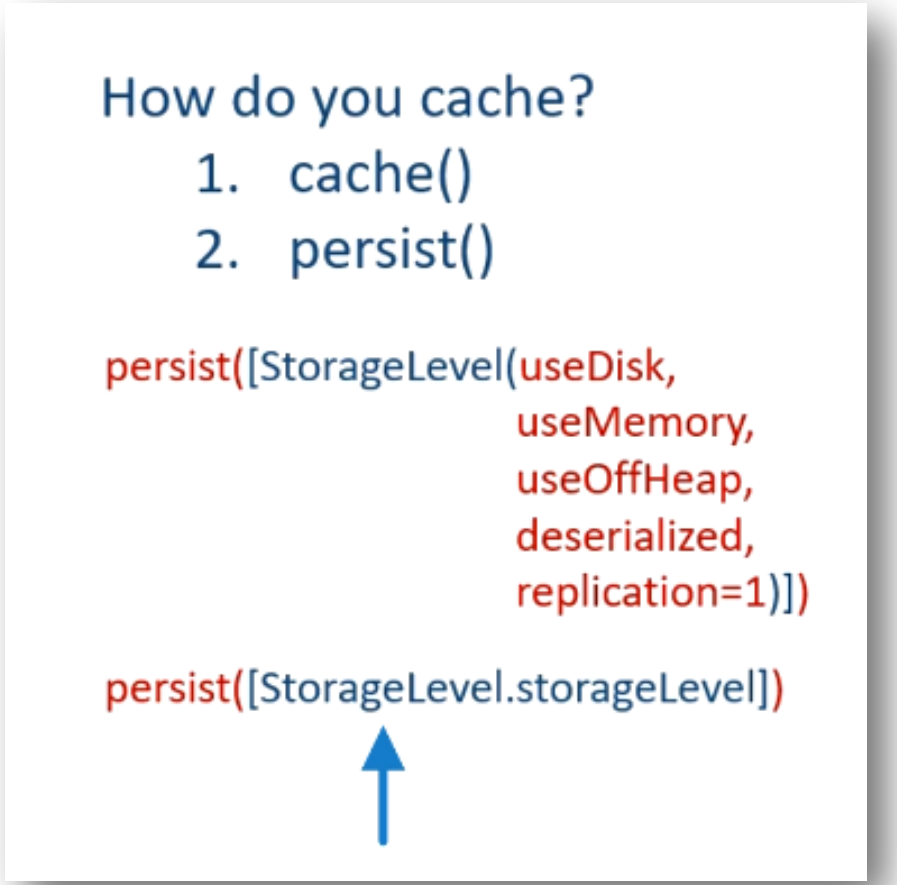
1. cache() method
2. persist() method

Let us start our discussion on this with the following question:

What is the difference between cache() and persist()?

Both of these methods will cache your Dataframe in the executor storage pool. At a high level, they are the same because they do the same thing - cache the Dataframe.

But there is a slight difference between cache() and persist().
The cache() method does not take any argument. However, the persist() method takes an optional argument for the storage level. So the signature for the persist method is like this shown below in the first code.
You also have an alternative for the persist method as highlighted below in the last code.



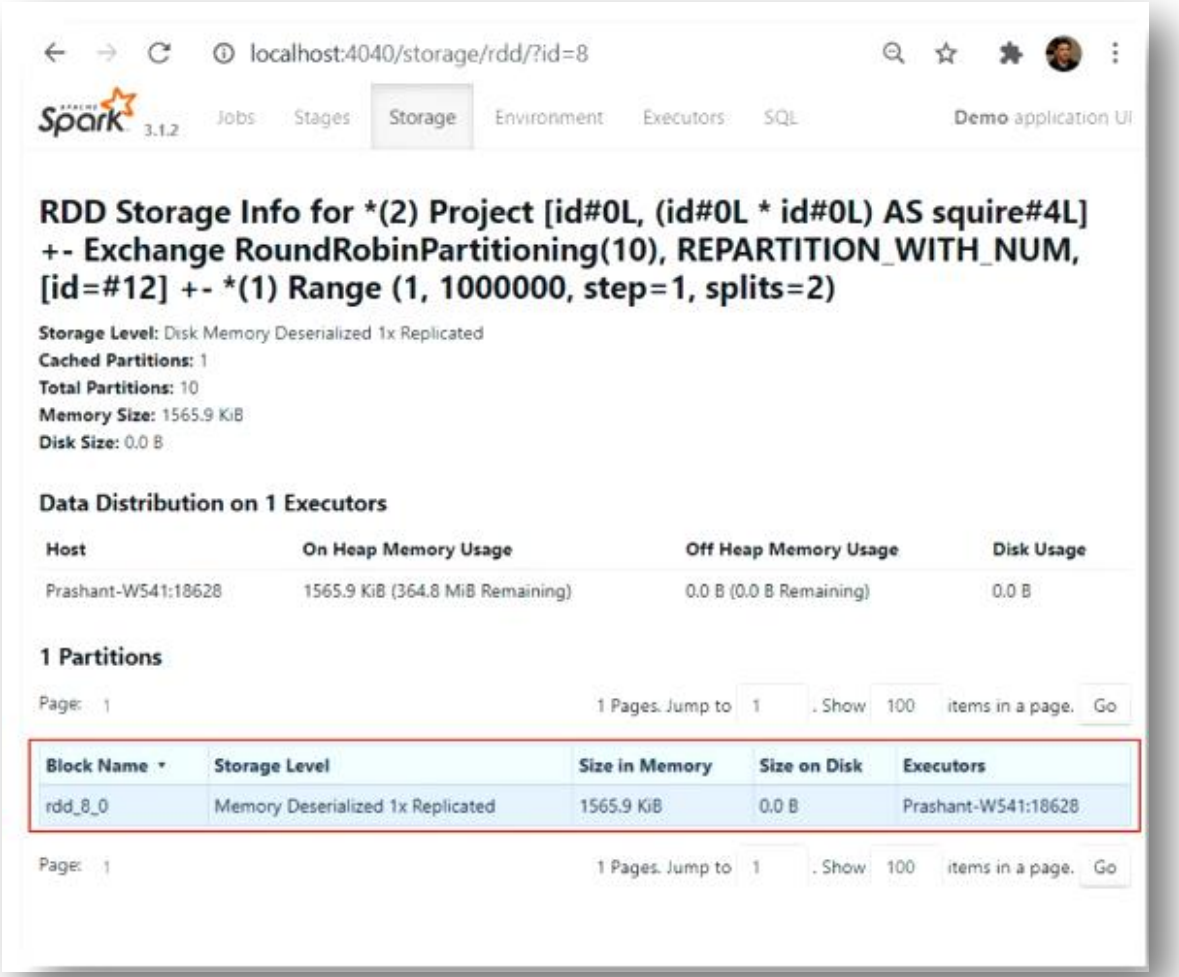
So, both of these alternatives are to specify the storage level.
So let me quickly recap the difference between cache() and persist()
The cache method does not take any argument, and it will cache your Dataframe using the default MEMORY_AND_DISK storage level.

However, they persist() method is more flexible.
The persist() method will also cache the Dataframe, but it allows you to configure the storage level.

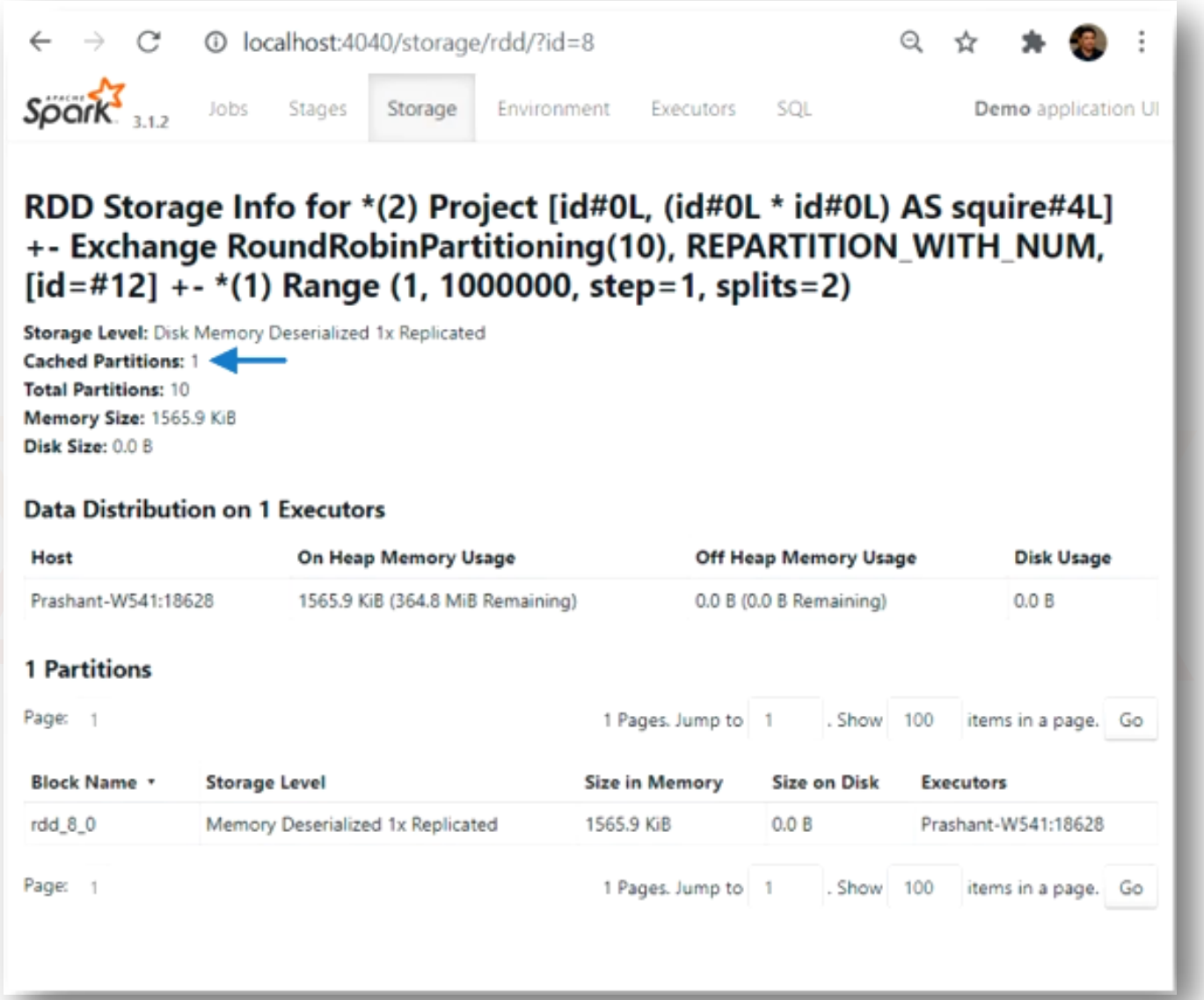
Now let's try to understand what is storage level with an example. We have a simple Dataframe code given below. I am creating a Dataframe of one million rows and repartitioning it to 10 uniform partitions. Why am I repartitioning? Because I wanted to demo you how a partitioned Dataframe is cached in your spark memory. If you are loading a Dataframe from parquet or other types of files, your Dataframe is most likely to have multiple partitions.
But I am creating a Dataframe at runtime, so I have to repartition it to show you how multiple partitions behave when you cache it.
Then I am calling the cache() method to cache my Dataframe in Spark storage memory.
Finally, I am executing the take(10) action.

```
df = spark.range(1, 1000000).toDF("id") \
          .repartition(10) \
          .withColumn("square", expr("id * id")) \
          .cache()

df.take(10)
```

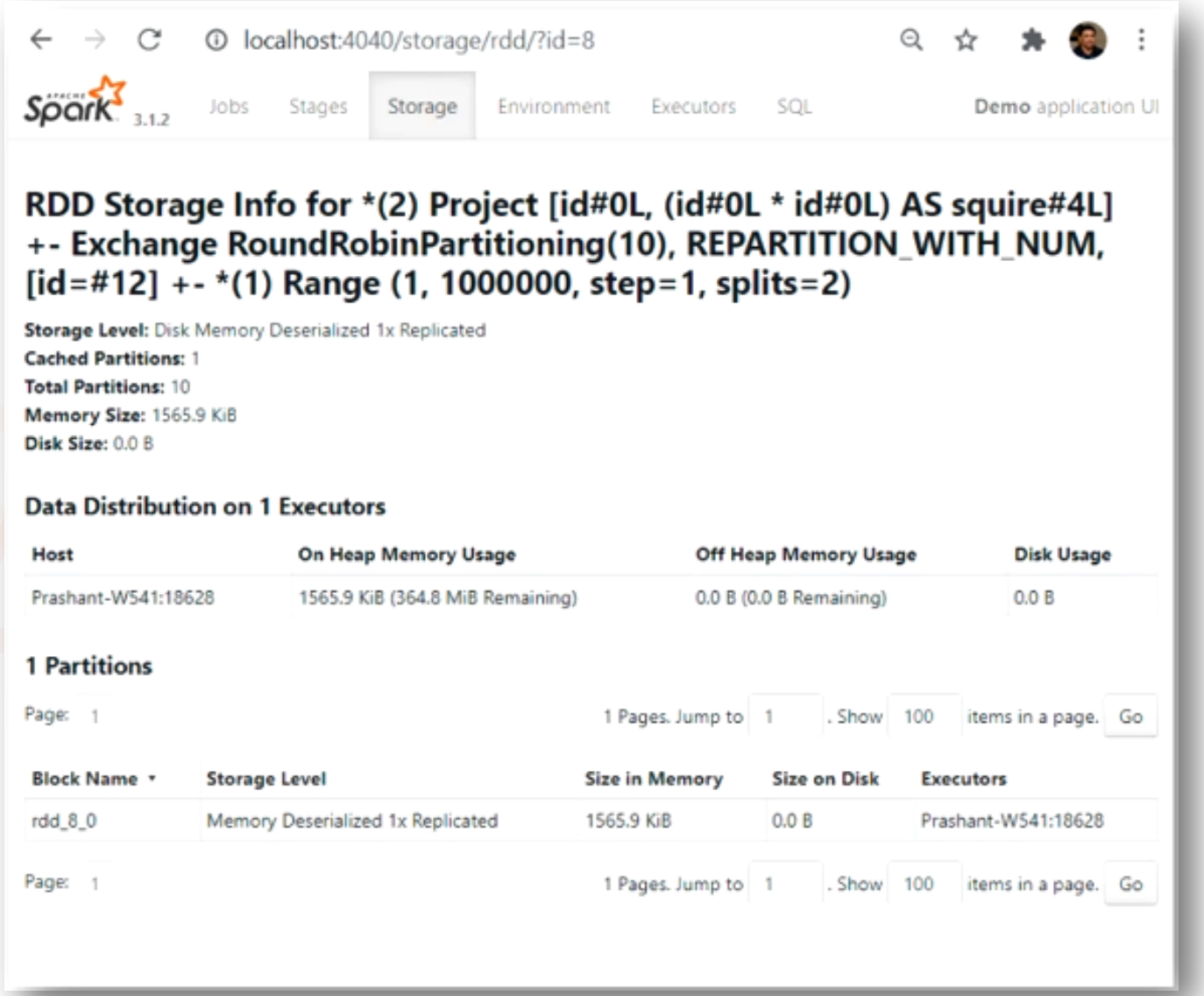

I ran this code and checked the storage tab of my Spark UI. Here is the screenshot for the same given below. You can see that I have one partition in the cache. This row shows the details of my cached partition. We have too many things here, let me explain it one by one. I created a Dataframe of 10 partitions.



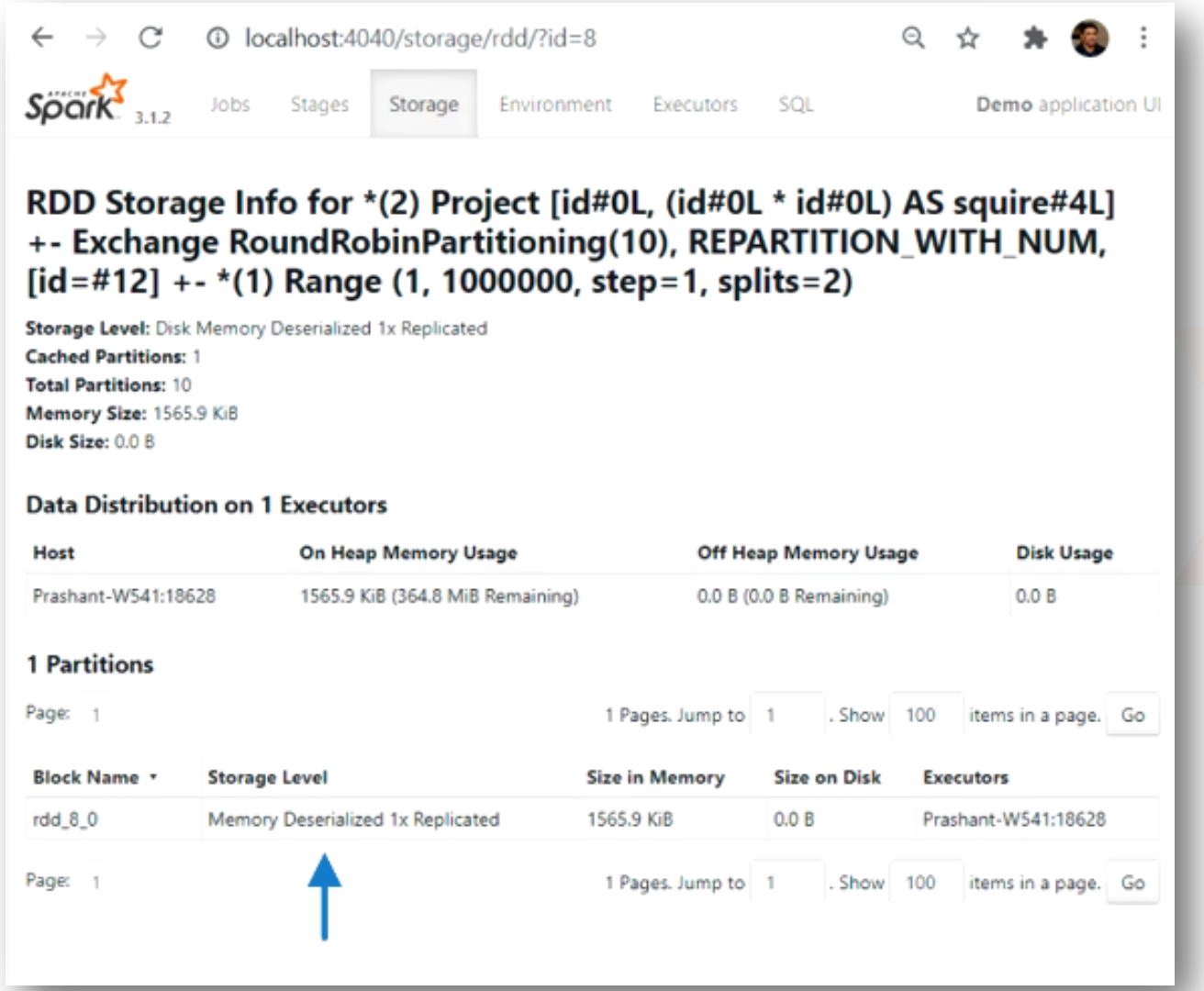
Then I cached it. But the spark is showing only one partition in the cache. This is because I used take(10) action. The take(10) action will bring only one partition in the memory and return one record from the loaded partition.
That's how take(10) is supposed to work.
Since we brought only one partition to memory, the spark will cache only one partition.



We also learned that Spark would cache the whole partition.
I am taking only ten records from the loaded partition, but Spark will cache the entire partition.
It means, spark will always cache the entire partition. If you have a memory to cache 7.5 partitions, Spark will cache only seven partitions because the 8th partition does not fit in the memory. Spark will never cache a portion of the partition. It will either cache the entire partition or nothing but never cache a portion of the partition.



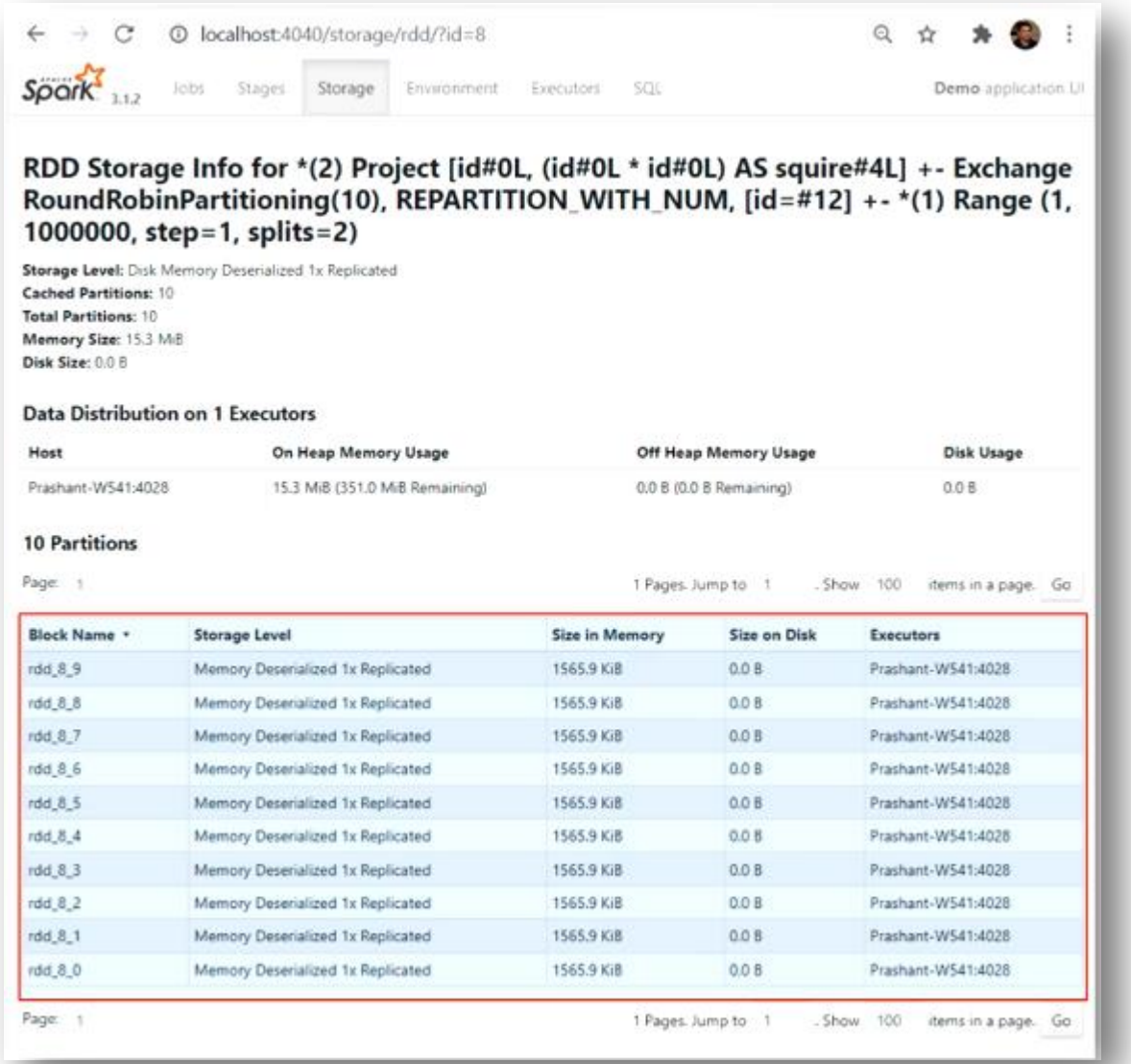
We also learned that the default storage level for the cache() method is Memory Deserialized 1 X replicated. And you can also see the size of the cached partition here next to the storage level.



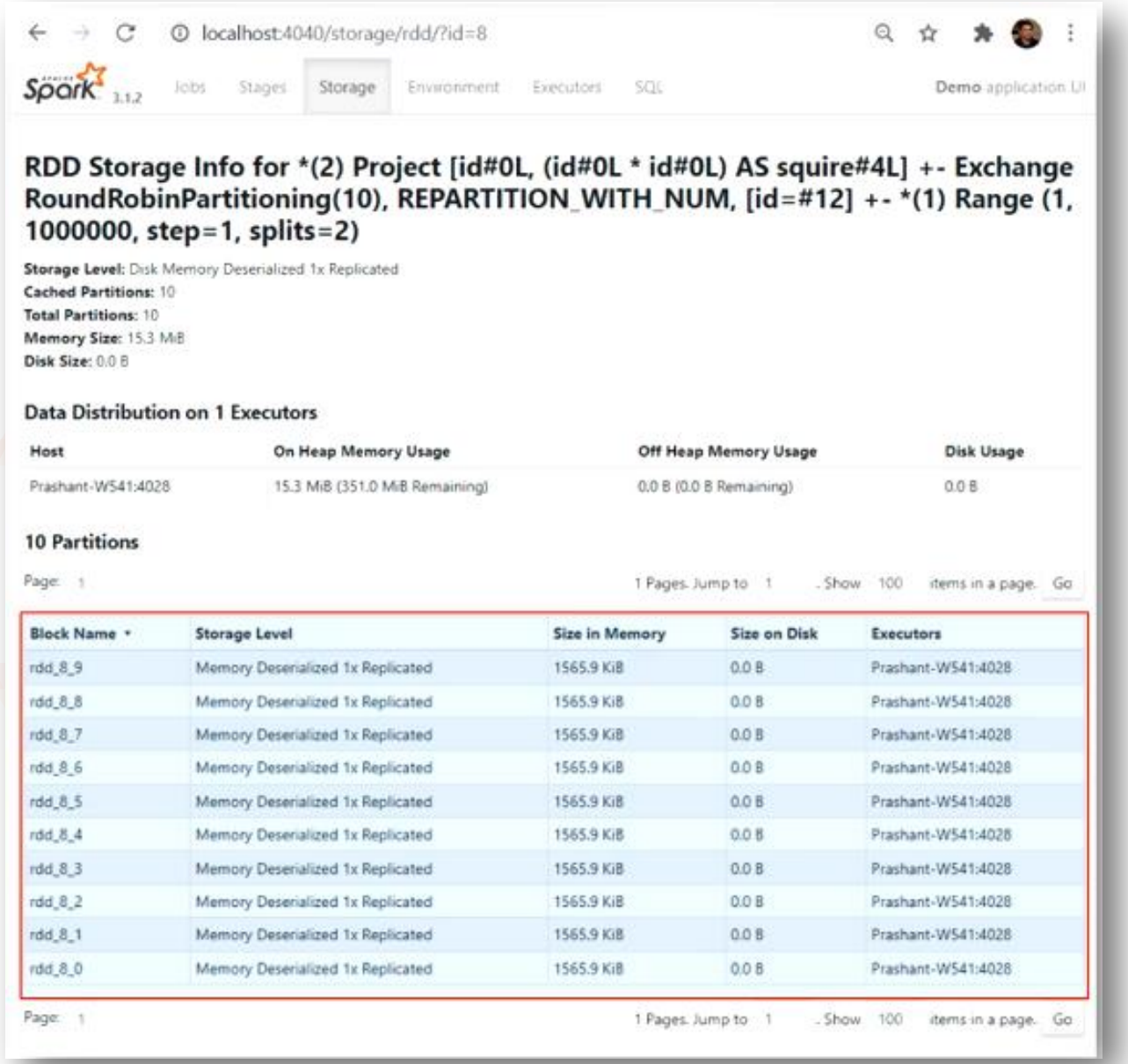
Now let me change the code and try it again.
I changed the take(10) method and replaced it with the count() action.
The count() action will bring all the partitions in the memory, compute the count and return it.

```
df = spark.range(1, 1000000).toDF("id") \
        .repartition(10) \
        .withColumn("square", expr("id * id")) \
        .cache()

df.count(    )
```


You can run this code and check the storage tab once again. And you will see the results as shown in the screenshot below.



Now you see all the ten partitions are cached().
Why? Because count() action forced all the partitions to be loaded in the executor memory.
We have enough memory to cache all ten partitions, so Spark is caching all the ten partitions.
And the storage level of all the cached blocks is still the same as Memory Deserialized 1 X replicated.



Now let's use the persist() method and change the storage level. But before that, let me summarize few things:

1. Spark cache() as well as persist() methods are lazy transformations. So they do not cache() anything until an action is executed.
2. The cache() and persist() methods are smart enough to cache only what you access. So if you are accessing only 3 or 5 partitions, Spark will cache only those partitions.
3. Spark will never cache a portion of a partition. It will always cache the whole partition if and only if the partition fits in the storage.

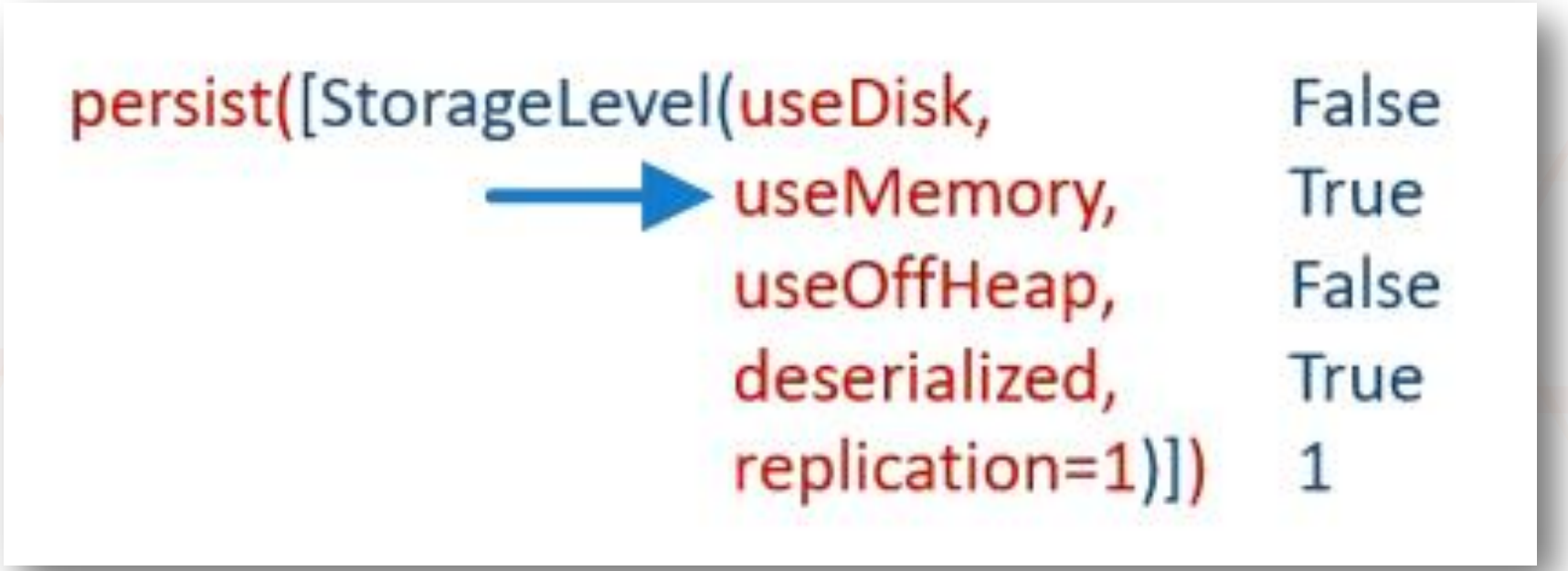
Here is my new code using the persist() method.
Everything is the same as earlier. But now, I am using persist method instead of the cache() method.

```
df = spark.range(1, 1000000).toDF("id") \
          .repartition(10) \
          .withColumn("square", expr("id * id")) \
          .persist(StorageLevel(False,
                                True,
                                False,
                                True,
                               1))
df.count(    )
```


Here is the format for the StorageLevel method.
And in the current example I am setting the corresponding values.

```
persist([StorageLevel(useDisk,          False
                      useMemory,        True
                      useOffHeap,       False
                      deserialized,     True
                      replication=1)])  1
```


I am saying use memory to cache the Spark Dataframe partitions.



I can also make the StorageLevel method like this.
So now I am extending the cache to the disk. If the partition does not fit into the Spark memory, Spark will cache it in the local disk of the executor.

```
persist([StorageLevel(useDisk,          True
                      useMemory,        True
                      useOffHeap,       False
                      deserialized,     True
                      replication=1)])  1
```


I can also extend it to off-heap memory for caching my Dataframe partitions. However, we already learned that you should have added some off-heap memory to Spark. If you haven't added off-heap memory for your Spark application, setting this parameter should not have any effect.

Now let's come to deserialization. What is this?
Spark always stores data on a disk in a serialized format. But when data comes to Spark memory, it must be deserialized into Java objects. That's mandatory for Spark to work correctly. But when you cache data in memory, you can cache it in serialized format or deserialized format. The deserialized format takes a little extra space, and the serialized format is compact. So you can save some memory using the serialized format. But when Spark deserializes the date wants to use it. So you will spend some extra CPU overhead.

You can cache your data in a serialized format and save some memory but incur CPU costs when Spark access the cached data.
Alternatively, you can cache it deserialized and save your CPU overhead.
The choice is yours, but the recommendation is to keep it deserialized and save your CPU.
And this configuration only applies to memory. Your disk-based cache is always serialized.

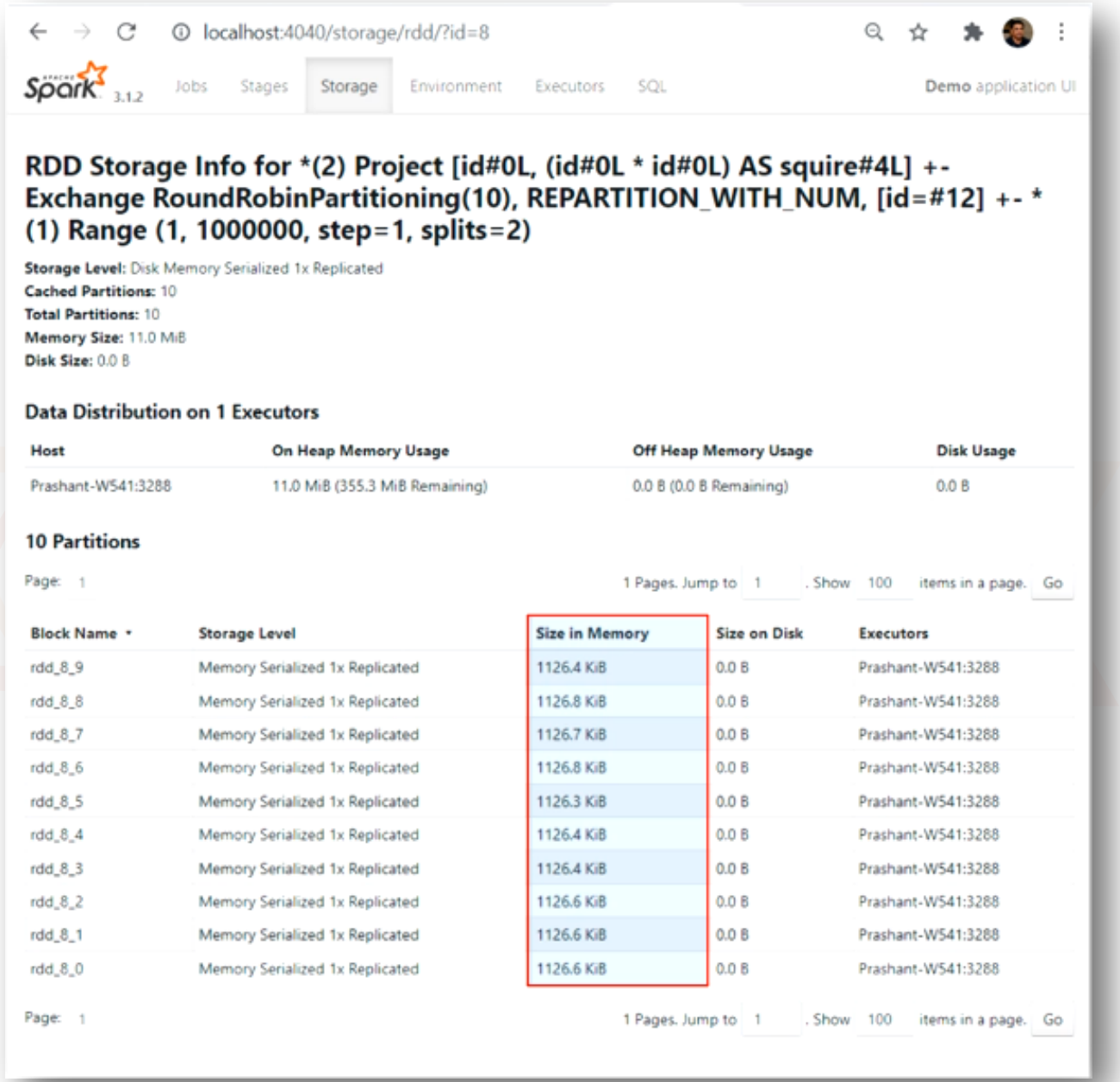
So I am changing the code and setting some different storage levels.
Here is my new code example given below.
This time, I am setting the storage level to cache data in memory. If I do not have enough memory, then cache it in the disk. I don't want to cache anything in off-heap memory because I haven't added off-heap memory for my application. I am also willing to try to cache my data in serialized format. So I am setting the deserialized to false.

```
df = spark.range(1, 1000000).toDF("id") \
          .repartition(10) \
          .withColumn("square", expr("id * id")) \
          .persist(StorageLevel(True,
                                True,
                                False,
                                False,
                                1))
df.count(    )
```


The last one is the replication factor.
I am setting it to one because I do not want to cache multiple copies of my data.
If you set this value to 3, Spark will cache three copies of your Dataframe partitions.
All those three copies will be cached on three different executors, but I think caching multiple copies is wastage of memory.
You can cache multiple copies if you have some specific reason.
Otherwise, one copy of cache is more than enough.

```
df = spark.range(1, 1000000).toDF("id") \
          .repartition(10) \
          .withColumn("square", expr("id * id")) \
          .persist(StorageLevel(True,
                                True,
                                False,
                                False,
                                1))
df.count(    )
```


I ran the code from the previous cell and saw the following in Spark UI.
I still see everything cached in memory.
And that's obvious because I have enough memory to cache all my partitions in the memory.
I enabled disk caching also, but Spark will cache it on the disk when I do not have enough memory.
So what I see here is perfectly fine.
This time I stored it as serialized. So you can see the size of each partition is smaller than earlier.
When I stored it as deserialized, it took more memory space, but the serialized storage takes a little lesser.



So, the persist method allows you to customize your cache storage level and cache it.
You can cache your data in memory only, or you can cache it in the disk only.
You can also configure it to store both in memory and disk.
If you have some off-heap memory, you can also expand your cache memory to off-heap storage. So assume you enabled all three storage levels, then Spark will cache your data in memory. That's the first preference. If you need more storage, Spark will use off-heap memory. You still want to cache more data, Spark will use the local disk.

```
persist([StorageLevel(useDisk,
                      useMemory,
                      useOffHeap,
                      deserialized,
                      replication=1)])
```

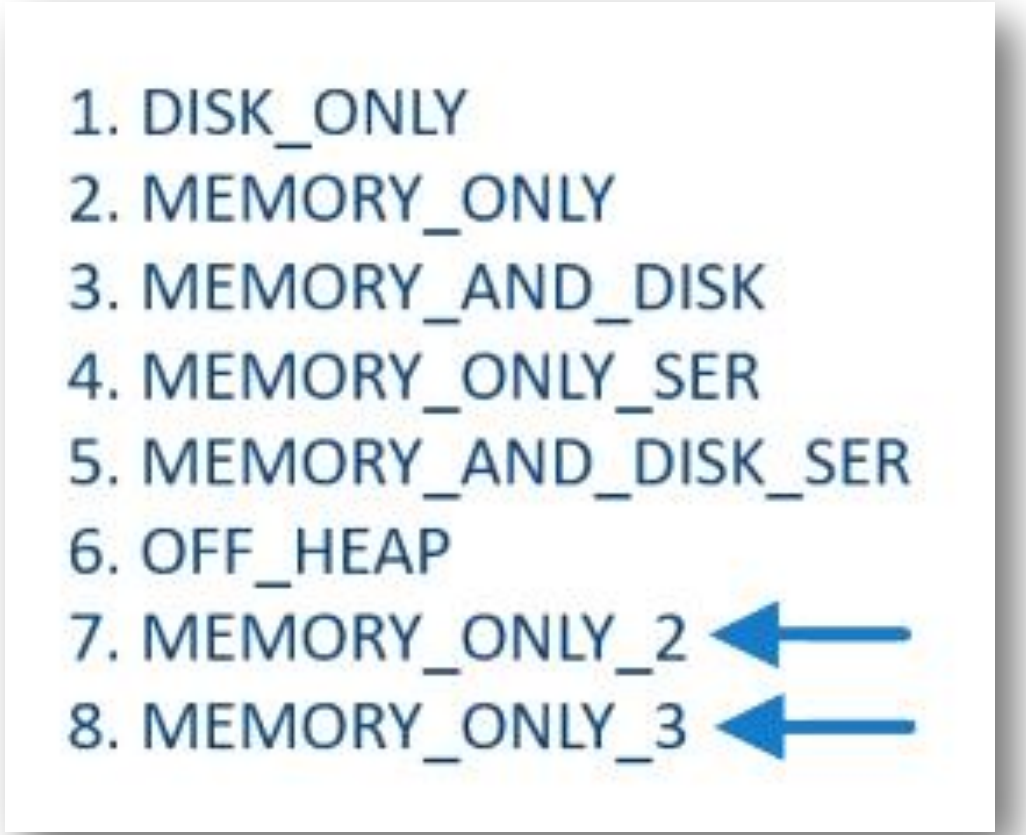

You can keep your cache in serialized or in deserialized formats.
But this configuration is only applicable for memory. Disk and off-heap do not support deserialized formats. They always store data in serialized formats.
The last option is to increase the replication factor of your cached blocks.
Increasing replication can give better data locality to Spark task schedular and make your application run faster. However, it will cost you a lot of memory.

```
persist([StorageLevel(useDisk,
                      useMemory,
                      useOffHeap,
                      deserialized,
                      replication=1)])
```


I used StorageLevel() function to demo the persist method.
But you can also use some predefined constants.
Here is an example shown below. In this example, I am using a predefined storage level constant.
The constant makes my code more readable, but you can also use the StorageLevel() function.

```
df = spark.range(1, 1000000).toDF("id") \
        .repartition(10) \
        .withColumn("square", expr("id * id")) \
        .persist(storageLevel=StorageLevel.MEMORY_AND_DISK)
df.count(10)
```

Here is an indicative list of available constants. You can refer to the documentation for a more exhaustive list.
The numbers 2 and 3 highlighted in these constants represent the replication factor.

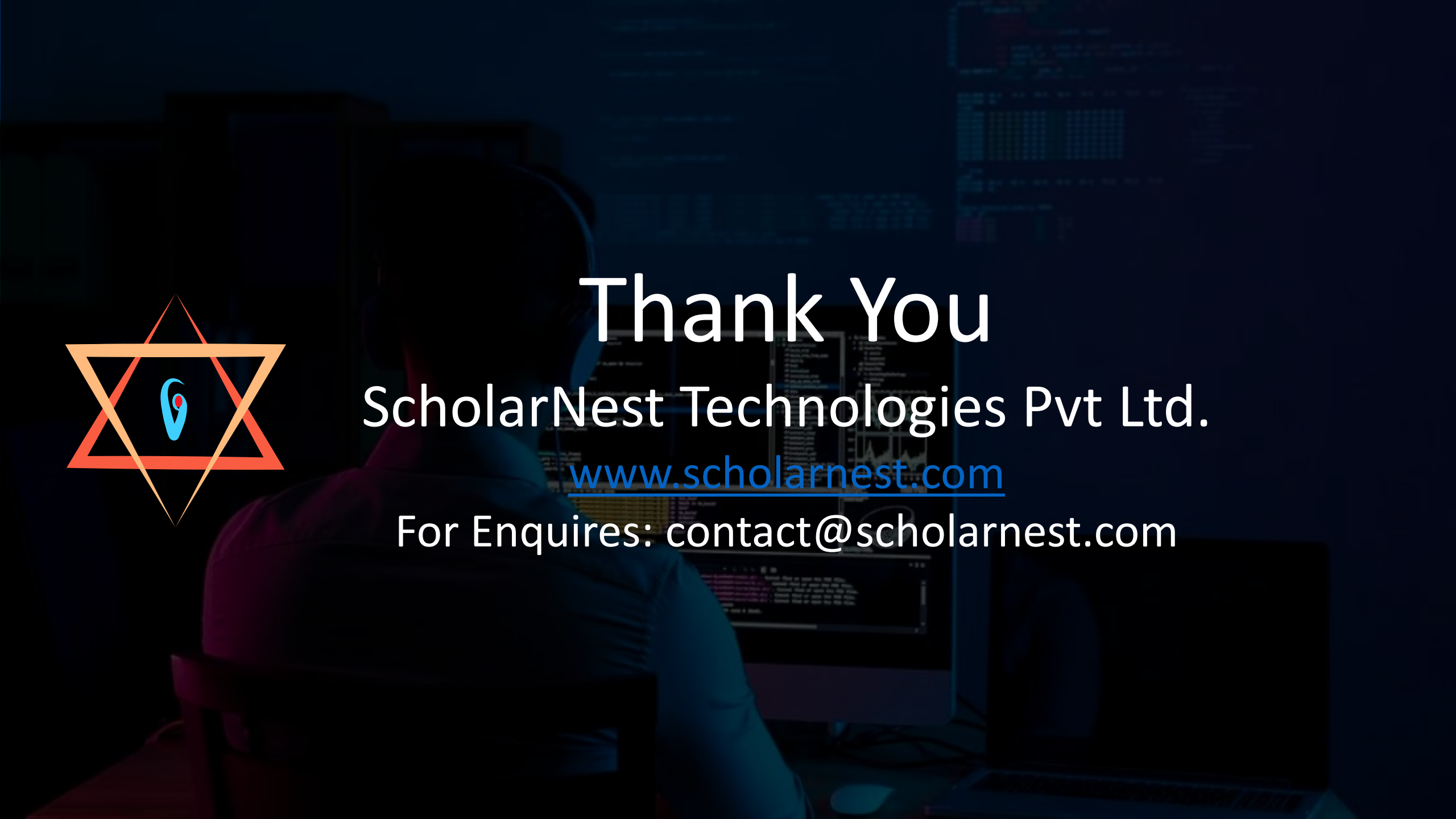


That's almost all except two questions we started this chapter with.
How to uncache?
When should we cache, and when should we avoid caching your data frames?
You can remove your date from the cache that you previously cached using the unpersist() method.
We do not have any method called uncache() but unpersist() will do the job.

1. How do you cache?
2. Why should you cache?
3. How to un-cache?
4. When cache and when not cache?
5. Caching formats?
6. Caching memory or disk?

Now the last question.
When should we cache, and when should we avoid caching your data frames?
When you want to access large Dataframe multiple times across Spark actions, You should consider caching your data frames. But make sure you know your memory requirements and configure your memory accordingly.
Do not cache your data frames when significant portions of them don't fit in the memory.
If you are not reusing your data frames frequently or too small, do not cache them.

1. How do you cache?
2. Why should you cache?
3. How to un-cache? df.unpersist()
4. When cache and when not cache?
5. Caching formats?
6. Caching memory or disk?



Thank You

ScholarNest Technologies Pvt Ltd.

www.scholarnest.com

For Enquires: contact@scholarnest.com

Spark Azure Databricks

Databricks Spark Certification and Beyond

Module:
Advanced Spark

Lecture:
Dataframe Hints

Data Frame Hints

In this chapter, I am going to talk about the hints in Spark.

Spark allows you to add two types of hints to your data frames and Spark SQL:

1. Partitioning Hints
2. Join Hints

We have the following four partitioning hints:

1. COALESCE – The COALESCE hint can be used to reduce the number of partitions to the specified number of partitions. It takes a partition number as a parameter.
2. REPARTITION – The REPARTITION hint can be used to repartition your Dataframe to the specified number of partitions. It takes a partition number, column names, or both as parameters.
3. REPARTITION_BY_RANGE – The REPARTITION_BY_RANGE is similar to REPARTITION, but it uses the data range for partitioning. We already learned about repartition and coalesce in an earlier video. These hints also offer similar functionality and behave in the same manner. But since these are hints, Spark doesn't guarantee that it will apply the hint.
4. REBALANCE – This one is added later than spark 3.0. The REBALANCE hint can be used to rebalance the query result output partitions so that every partition is of a reasonable size (not too small and not too big). It can take column names as parameters and try its best to partition the query result by these columns. This is the best effort: Spark will split the skewed partitions to make these partitions not too big if there are skews.

We also have four join hints:

1. BROADCAST alias BROADCASTJOIN and MAPJOIN
2. MERGE alias SHUFFLE_MERGE and MERGEJOIN
3. SHUFFLE_HASH
4. SHUFFLE_REPLICATE_NL

These hints allow you to suggest the join strategy for Spark.
When different join strategy hints are specified on both sides of a join, Spark prioritizes hints in the same order as I have listed these names.
The last one is known as shuffle-and-replicate nested loop join.
These join hints also have some alias. So the BROADCAST and BROADCASTJOIN and MAPJOIN are the same.
They are just alias for the same hint.

Now let us see how to use these hints. If you are using Spark SQL, you can use these hints using the following syntax.
And there are some examples for the same highlighted below.

Using Hints in Spark SQL

```
/*+ hint [ , ... ] */
```

```
SELECT /*+ COALESCE(3) */ * FROM t;

SELECT /*+ REPARTITION(3) */ * FROM t;

SELECT /*+ BROADCAST(t1) */ * FROM t1
INNER JOIN t2 ON t1.key = t2.key;

SELECT /*+ MERGE(t1) */ * FROM t1 INNER
JOIN t2 ON t1.key = t2.key;
```


So using join hints in Spark SQL is simple and straight. But how do we apply them in Dataframes?

We have two methods to apply hints in a Dataframe:

1. Use spark sql functions
2. Use Dataframe.hint() method

I have an example here that uses both types of hints in a Dataframe.
(Reference: https://github.com/ScholarNest/PySpark-Examples/tree/main/dataframe-hints)

```python
from pyspark.sql import SparkSession
from pyspark.sql.functions import *

if __name__ == "__main__":
    spark = SparkSession \
        .builder \
        .appName("Demo") \
        .master("local[3]") \
        .getOrCreate()

    flight_time_df1 = spark.read.json("data/d1/")
    flight_time_df2 = spark.read.json("data/d2/")

    # join_df = flight_time_df1.join(flight_time_df2.hint("broadcast"), "id", "inner")

    join_df = flight_time_df1.join(broadcast(flight_time_df2), "id", "inner") \
        .hint("COALESCE", 5)

    join_df.show()
    print("Number of Output Partitions:" + str(join_df.rdd.getNumPartitions()))

```


So, I am creating two Dataframe flight_time_df1 and flight_time_df2. The df1 is a large Dataframe, and the df2 is a small one.

```
1  from pyspark.sql import SparkSession
2  from pyspark.sql.functions import *
3
4  if __name__ == "__main__":
5      spark = SparkSession \
6          .builder \
7          .appName("Demo") \
8          .master("local[3]") \
9          .getOrCreate()
10
11     flight_time_df1 = spark.read.json("data/d1/")
12     flight_time_df2 = spark.read.json("data/d2/")
13
14     # join_df = flight_time_df1.join(flight_time_df2.hint("broadcast"), "id", "inner")
15
16     join_df = flight_time_df1.join(broadcast(flight_time_df2), "id", "inner") \
17         .hint("COALESCE", 5)
18
19     join_df.show()
20     print("Number of Output Partitions:" + str(join_df.rdd.getNumPartitions()))
21
```

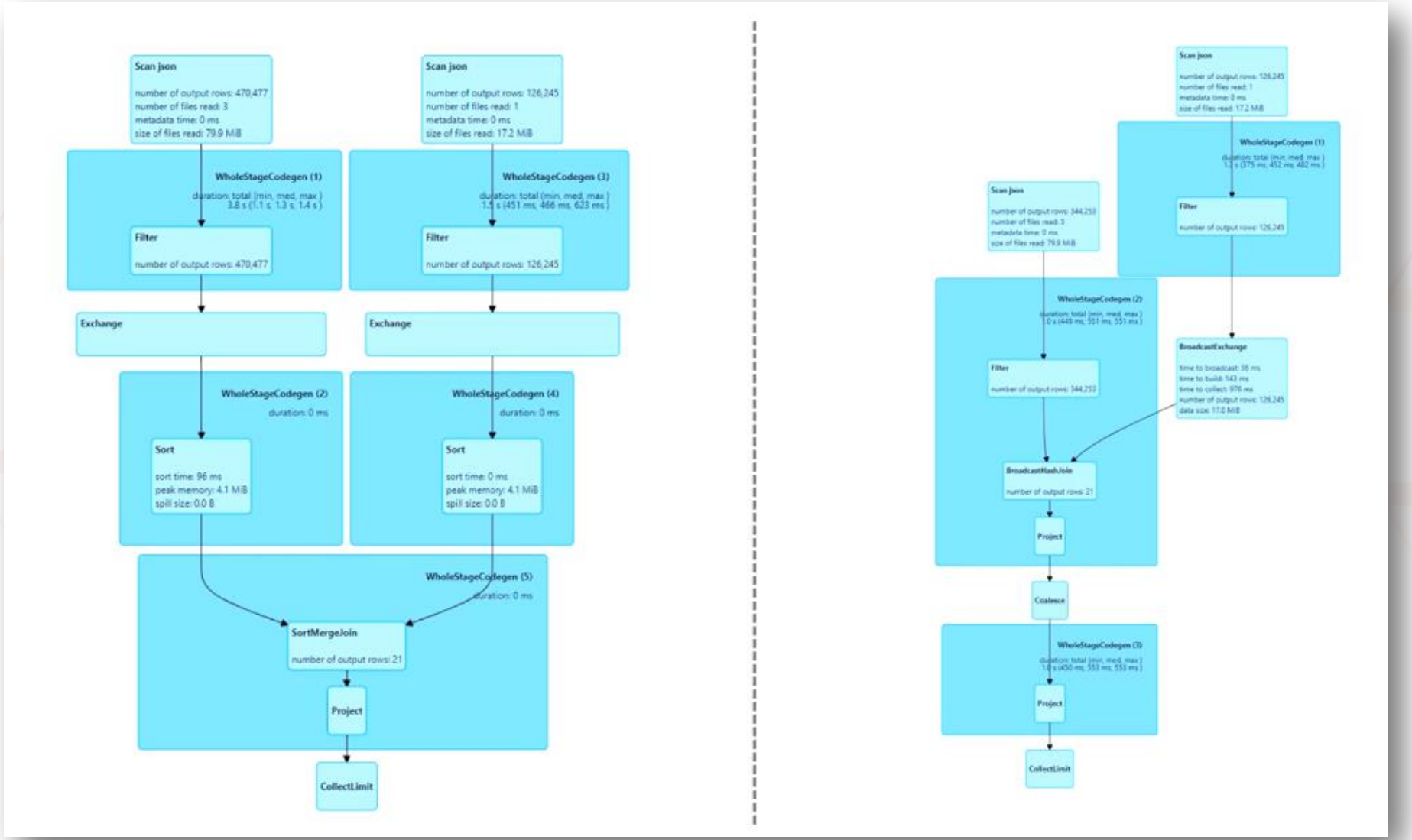

So I wanted to apply the broadcast hint to the df2 so I can avoid shuffle sort. And that's why I applied a broadcast() function around the flight_time_df2. The broadcast() function is one approach to apply the broadcast hint to a Dataframe. I could have also done it like this.

```
1  from pyspark.sql import SparkSession
2  from pyspark.sql.functions import *
3
4  if __name__ == "__main__":
5      spark = SparkSession \
6          .builder \
7          .appName("Demo") \
8          .master("local[3]") \
9          .getOrCreate()
10
11     flight_time_df1 = spark.read.json("data/d1/")
12     flight_time_df2 = spark.read.json("data/d2/")
13
14     # join_df = flight_time_df1.join(flight_time_df2.hint("broadcast"), "id", "inner")
15
16     join_df = flight_time_df1.join(broadcast(flight_time_df2), "id", "inner") \
17         .hint("COALESCE", 5)
18
19     join_df.show()
20     print("Number of Output Partitions:" + str(join_df.rdd.getNumPartitions()))
21
```

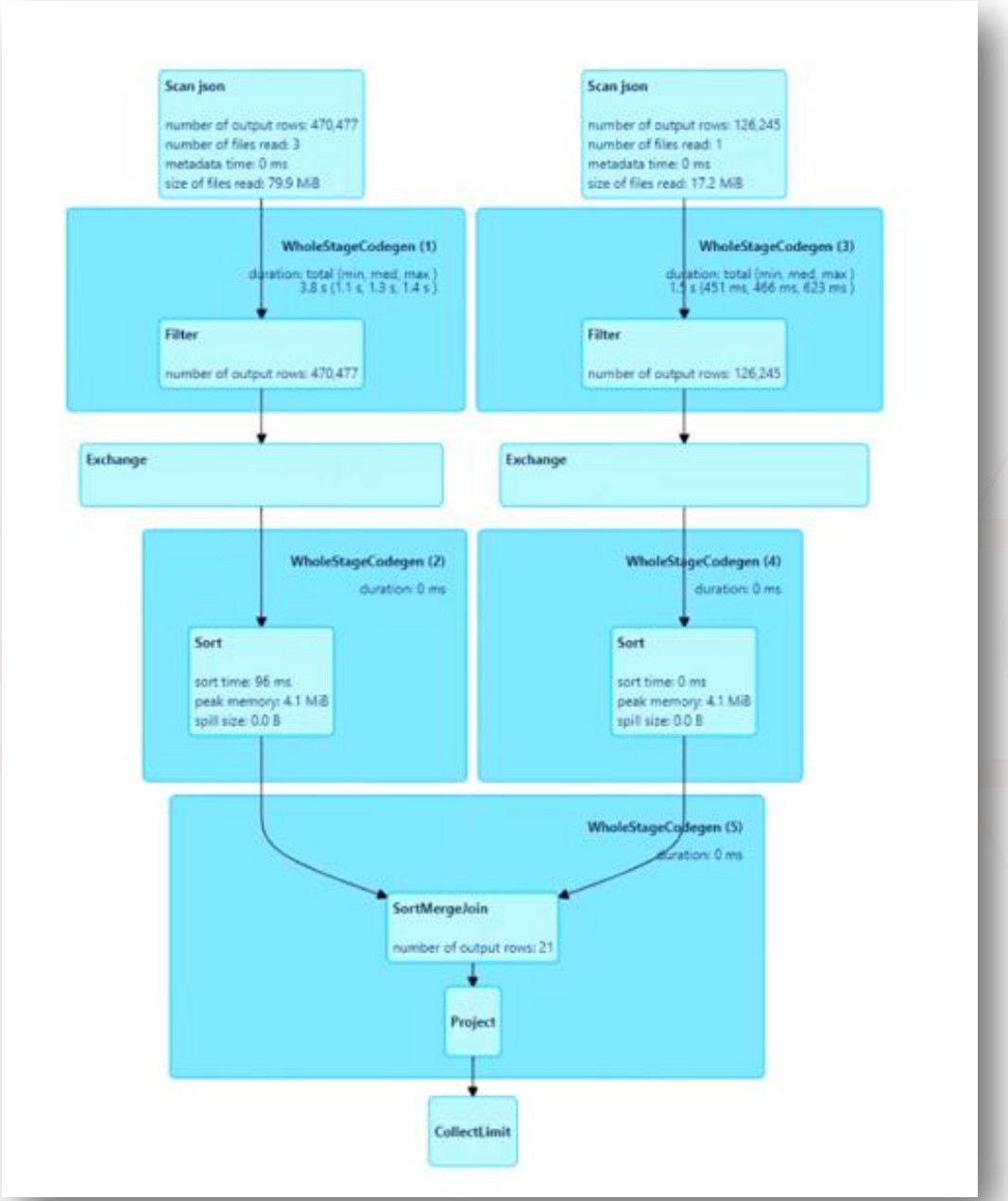

Both the methods have the same effect.
So I am joining two data frames and applying a coalesce hint to the join result. I know the default shuffle partition value is set to 200. So If I do not apply the coalesce hint, the join_df will have 200 partitions. But I applied a coalesce(5) hint. So Spark will coalesce it into five partitions only.

```
1  from pyspark.sql import SparkSession
2  from pyspark.sql.functions import *
3
4  if __name__ == "__main__":
5      spark = SparkSession \
6          .builder \
7          .appName("Demo") \
8          .master("local[3]") \
9          .getOrCreate()
10
11     flight_time_df1 = spark.read.json("data/d1/")
12     flight_time_df2 = spark.read.json("data/d2/")
13
14     # join_df = flight_time_df1.join(flight_time_df2.hint("broadcast"), "id", "inner")
15
16     join_df = flight_time_df1.join(broadcast(flight_time_df2), "id", "inner") \
17         .hint("COALESCE", 5)
18
19     join_df.show()
20     print("Number of Output Partitions:" + str(join_df.rdd.getNumPartitions()))
21
```

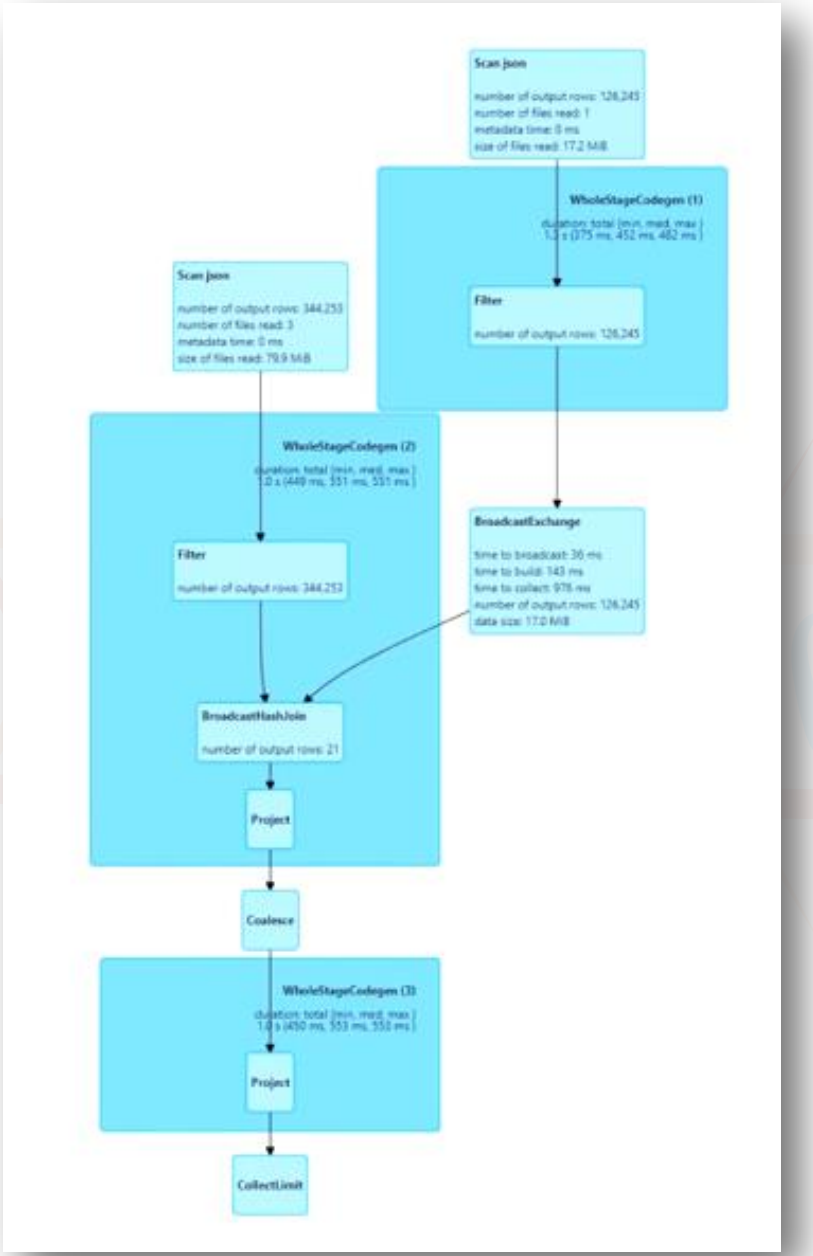

I executed the previous code without these two hints and also executed them with both the hints. Here are two execution plans.

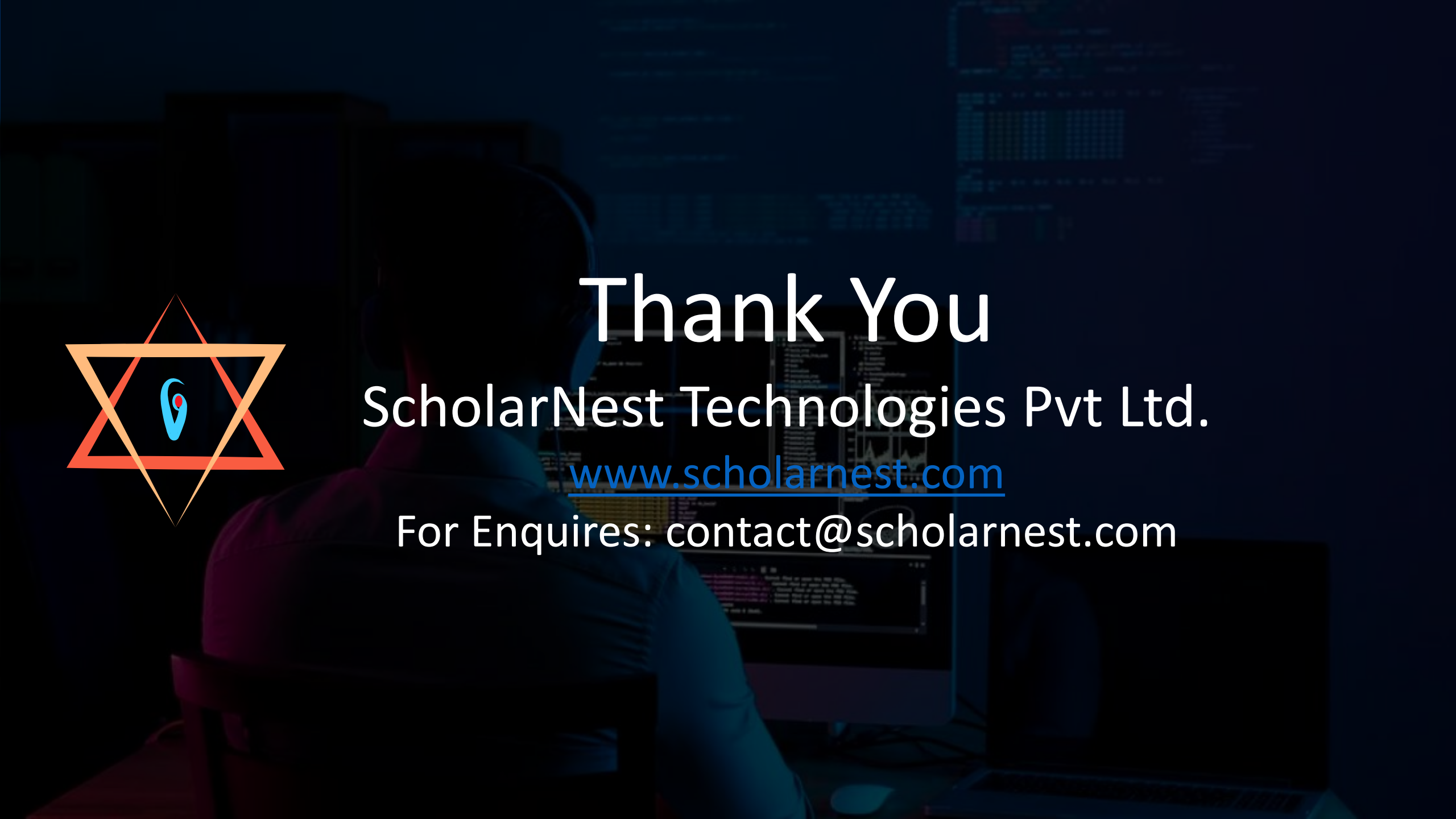


This plan is for without the hints. You can see that the this plan applies shuffle sort and then a sort-merge join.



This plan is for both the hints.
This plan applies a broadcast join alias also applies a coalesce in the end.





Thank You

ScholarNest Technologies Pvt Ltd.

www.scholarnest.com

For Enquires: contact@scholarnest.com

Spark Azure Databricks

Databricks Spark Certification and Beyond

Module:
Advanced Spark

Lecture:
Repartition Vs Coalesce

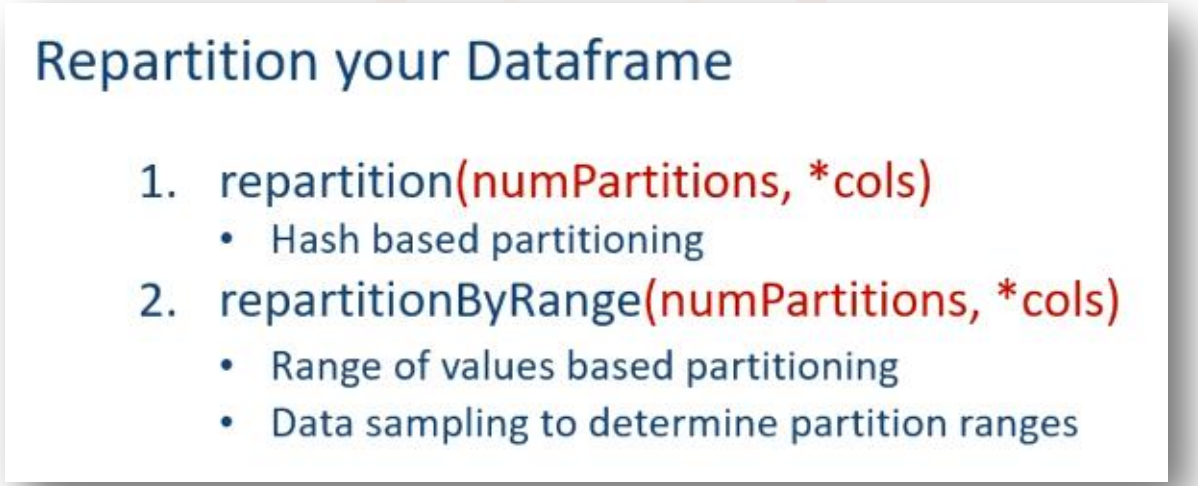
Repartition Vs Coalesce

Now I want to cover the following essential concepts:

1. Repartition and Coalesce
2. Dataframe Hints
3. Broadcast
4. Speculative Execution
5. Spark Schedulers

Let's start with the Dataframe repartition, and then I will talk about the Coalesce.

Spark Dataframe API allows you to repartition your Dataframe. They offer you two methods as listed below. Both the methods are almost the same except for one difference. The repartition() method will use a hash function to determine the target partition of your row. However, the repartitionByRange() will partition your data using a range of values. For example, 0 to 10 goes in the first partition, then 10 to 20 goes in the second partition, and so on. That's the only difference. Otherwise, both the functions work in the same way. You should also understand one more important thing about repartitionByRange(). The repartitionByRange() method internally uses data sampling to estimate the partition ranges. So the output of repartitionByRange() may not be consistent. I mean, if you execute the same code twice, you may see different partition ranges. And that should be perfectly fine in many cases.



Now let's see how do they work. The repartition() method takes two optional arguments:

1. Number of partitions
2. List of columns

One of these two arguments is mandatory. So all the following examples are valid.

repartition(numPartitions, *cols)

Examples

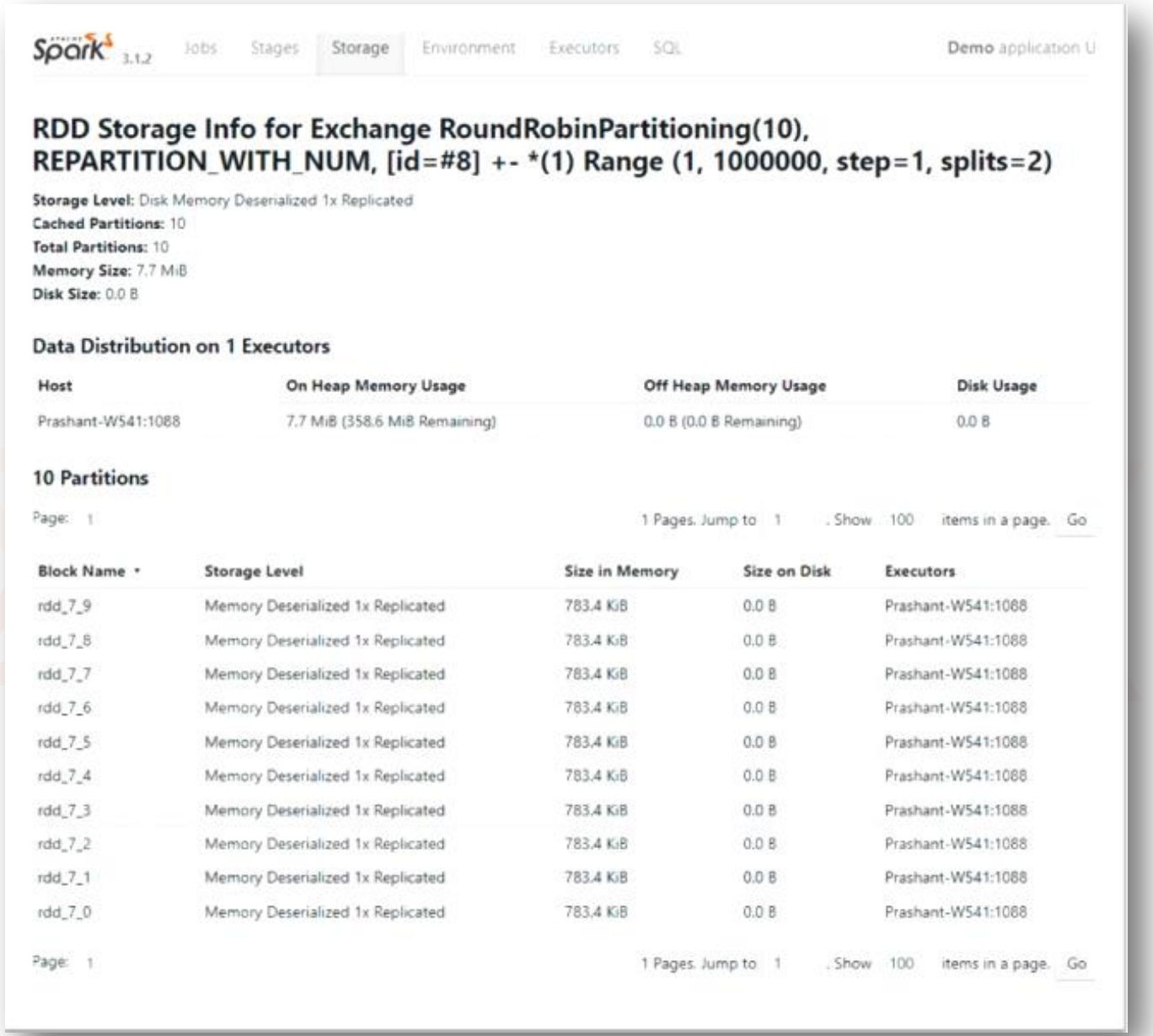
1. repartition(10)
2. repartition(10, "age")
3. repartition(10, "age", "gender")
4. repartition("age")
5. repartition("age", "gender")

Similarly, you can also assume repartitionByRange() because both have the same structure. Now let's see how they work.

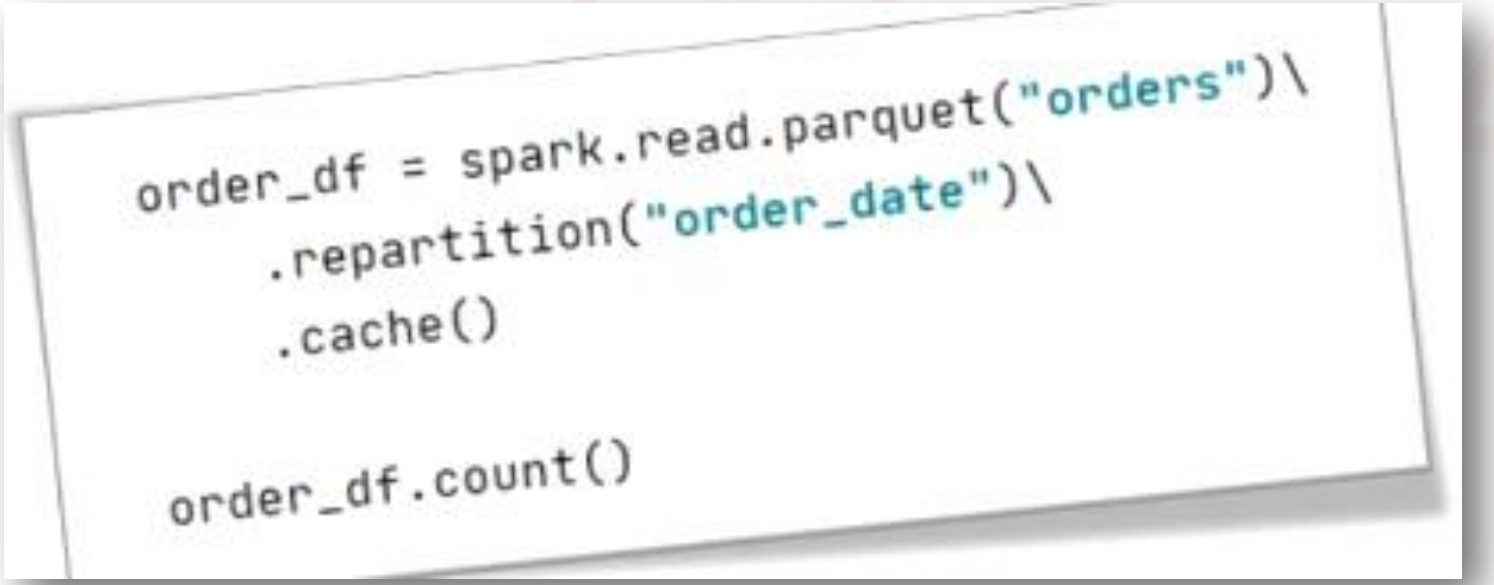
The repartition(10) will create ten new partitions of your Dataframe. How? Simple! The repartition() is a wide dependency transformation. So it will cause a shuffle/sort of your Dataframe, and the final output will have ten new partitions. And all those ten partitions will be uniform in size. Here is an example code shown below.

So what am I doing here? I am creating a Dataframe of one million rows, repartition it to 10 uniform partitions, and cache it. Why cache it? Because I want to see them in Spark UI and learn how repartition behaves.

So what do you see on the Spark UI?
We have ten partitions, and all are of the same size.
So the repartition(10) creates ten uniform partitions.



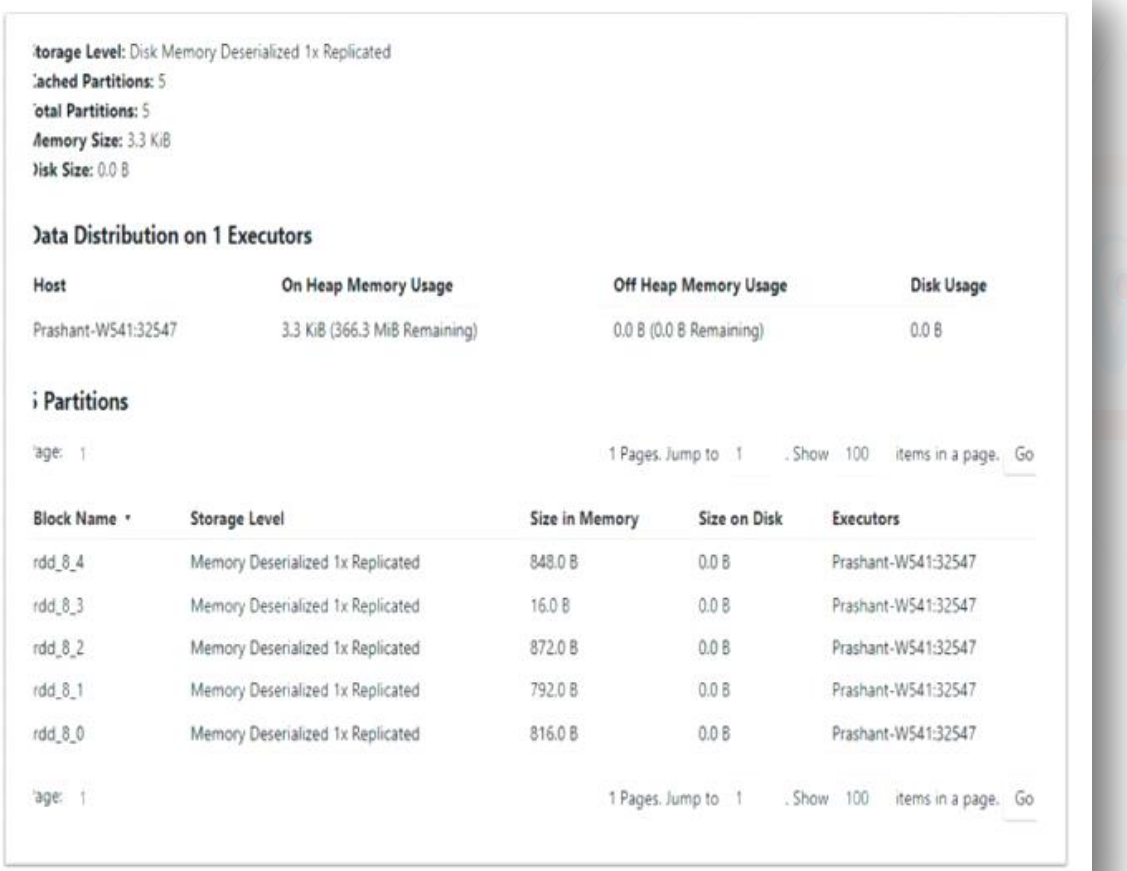
Here is another example to see how repartition works for a column.
What am I doing in this new code? I am reading a parquet data source and repartitioning it using order_date. And how many partitions do I get? I got ten partitions once again.
But How? Why I got ten partitions, and why not five partitions or maybe 200 partitions?
The repartition() is a wide dependency transformation. So it will cause shuffle/sort and repartition of your data using the order_date column. But the number of partitions is controlled by the spark.sql.shuffle.partitions configuration. I configured the spark.sql.shuffle.partitions to 10, so I got ten partitions.



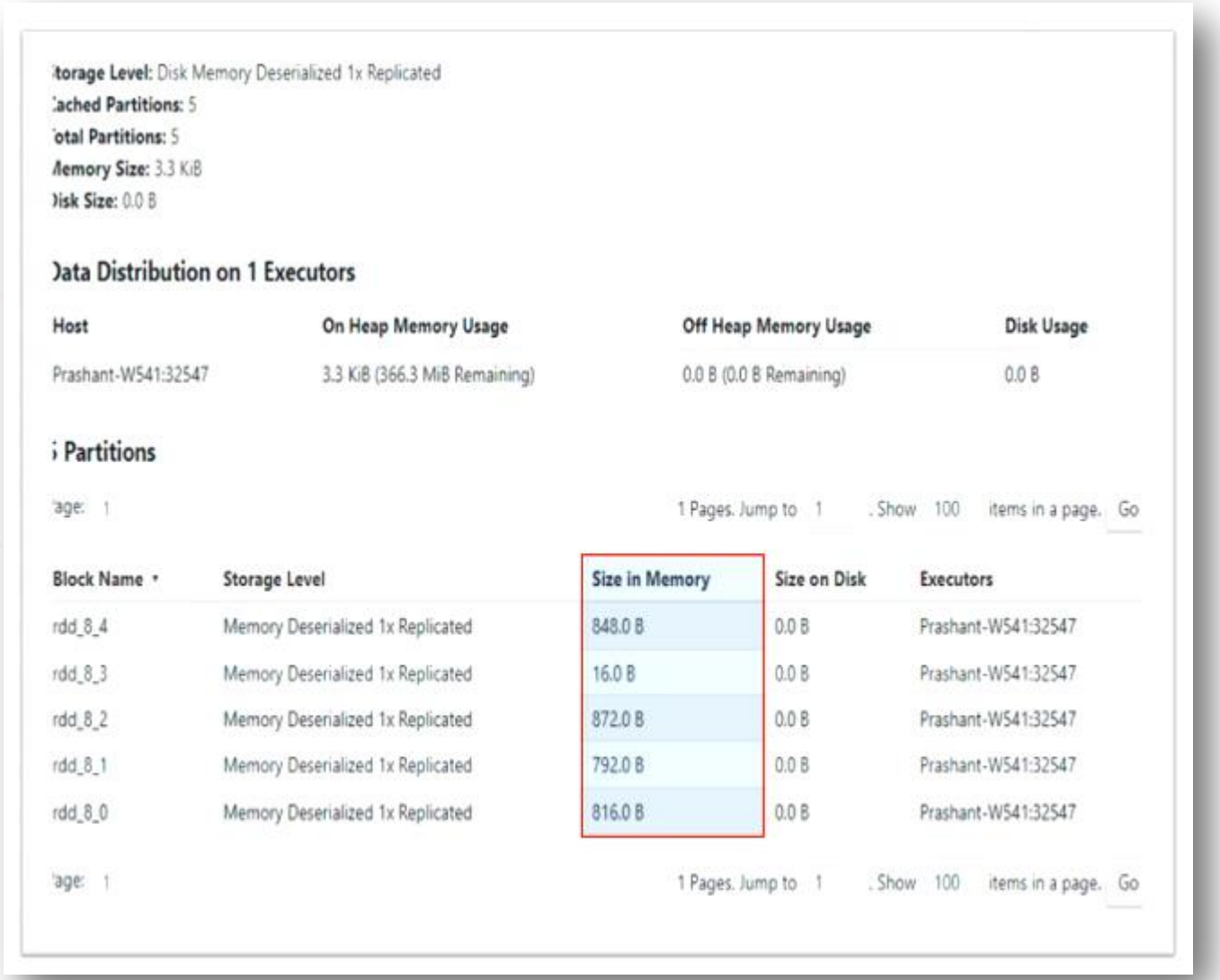
But you can override this behaviour by supplying the number of partitions. Here is another example.

This time, I want only five output partitions, and I also want those partitions based on a column name. So Spark will partition my data on the order_date and limit the number of output partitions to five.

```
order_df = spark.read.parquet("orders")\
    .repartition(5, "order_date")\
    .cache()

order_df.count()
```



But you can also see that the partition size is not uniform.
So repartition on a column name does not guarantee a uniform partitioning.



torage Level: Disk Memory Deserialized 1x Replicated
ached Partitions: 5
otal Partitions: 5
lemory Size: 3.3 KiB
)isk Size: 0.0 B

)ata Distribution on 1 Executors

Host	On Heap Memory Usage	Off Heap Memory Usage	Disk Usage
Prashant-W541:32547	3.3 KiB (366.3 MiB Remaining)	0.0 B (0.0 B Remaining)	0.0 B

; Partitions

'age: 1 1 Pages. Jump to 1 . Show 100 items in a page. Go

Block Name	Storage Level	Size in Memory	Size on Disk	Executors
rdd_8_4	Memory Deserialized 1x Replicated	848.0 B	0.0 B	Prashant-W541:32547
rdd_8_3	Memory Deserialized 1x Replicated	16.0 B	0.0 B	Prashant-W541:32547
rdd_8_2	Memory Deserialized 1x Replicated	872.0 B	0.0 B	Prashant-W541:32547
rdd_8_1	Memory Deserialized 1x Replicated	792.0 B	0.0 B	Prashant-W541:32547
rdd_8_0	Memory Deserialized 1x Replicated	816.0 B	0.0 B	Prashant-W541:32547

'age: 1 1 Pages. Jump to 1 . Show 100 items in a page. Go

So let me summarize few points:

1. You can use repartition() to create uniform partitions of your Dataframe.
2. You can also use one or more columns to repartition your data frames.
3. Repartition causes a shuffle/sort, and the number of output partitions depends on the shuffle partitions configuration.
4. You can override the shuffle partition configuration by setting the numPartitions argument.
5. Repartitioning on column name does not ensure the uniform size of output partitions.

That's all about repartition except for one last question:

When do you want to repartition your Dataframe?

Repartition will cause a shuffle/sort, which is an expensive operation.

You should almost always try to avoid unnecessary shuffle/sort.

So repartitioning your Dataframe must be done with caution.

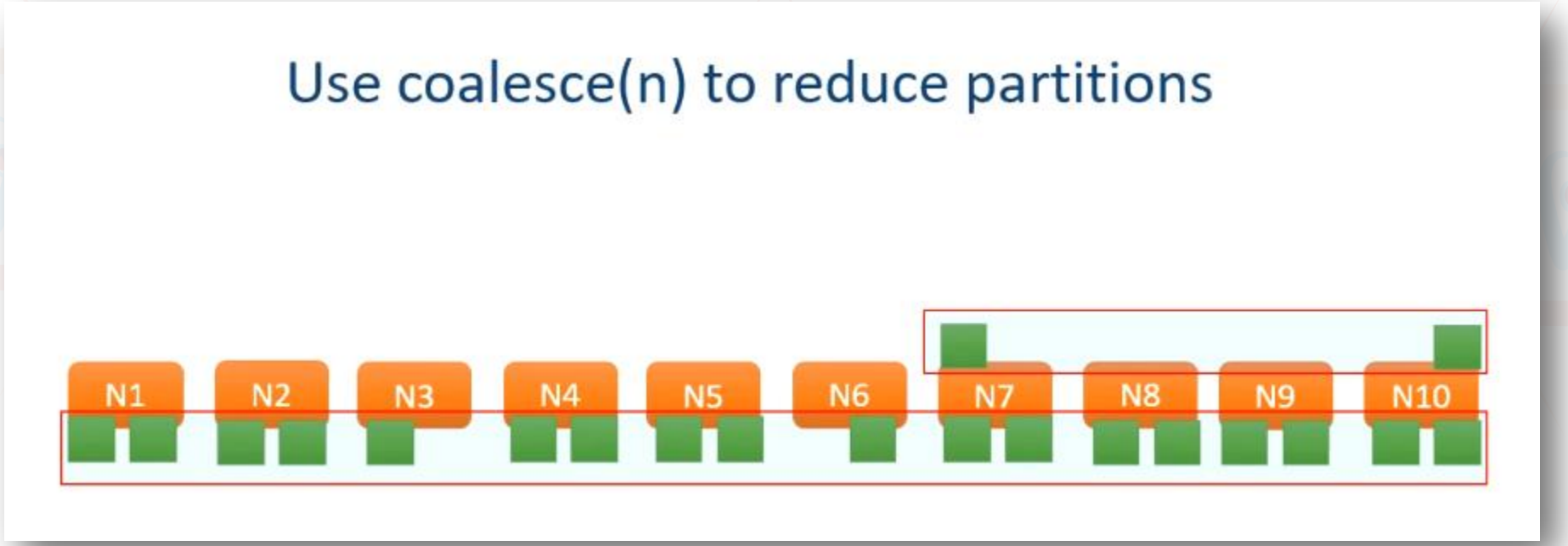
If you do not see any benefit of repartitioning, you must avoid it.

That's all about repartitioning.
You should use repartitioning when you want to increase the number of partitions or repartition on specific columns.
We should not use repartitioning to reduce the number of partitions.
Instead, you can use the Coalesce method for reducing the number of partitions.
Here is the structure of the coalesce method.

coalesce(numPartitions)

The coalesce() method takes the target number of partitions and combines the local partitions available on the same worker node to achieve the target.

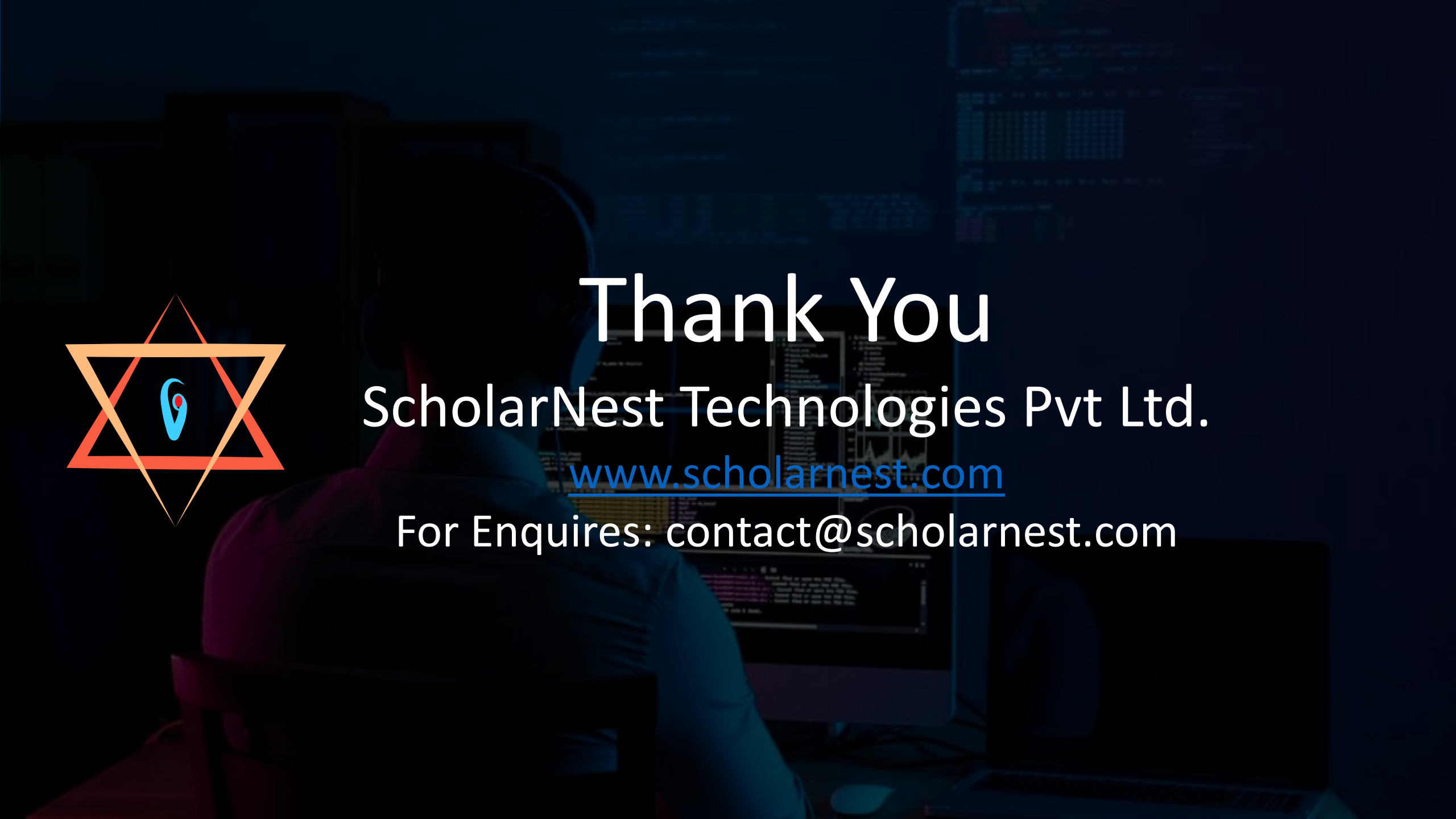
For example, let's assume you have a ten-node cluster.
And you have a Dataframe of 20 partitions.
Those 20 partitions are spread on these ten worker nodes.



Now you want to reduce the number of partitions to 10.
So you executed coalesce(10) on your Dataframe.
So Spark will try to collapse the local partitions and reduce your partition count to 10.
The final state might look like this.

You must learn the following things about the coalesce:

1. Coalesce doesn't cause a shuffle/sort. It will combine local partitions only. So you should use coalesce when you want to reduce the number of partitions.
2. If you try to increase the number of partitions using coalesce, it will do nothing. You must use repartition() to increasing the number of partitions. And you should also avoid using repartition to reduce the number of partitions. Why? Because you can reduce your partitions using coalesce without doing a shuffle? You can also reduce your partition count using repartition(), but it will cost you a shuffle operation.
3. Finally, coalesce can cause skewed partitions. So try to avoid drastically decreasing the number of partitions. It can cause skewed partitions, which may lead to an OOM exception.



Thank You

ScholarNest Technologies Pvt Ltd.

www.scholarnest.com

For Enquires: contact@scholarnest.com

Spark Azure Databricks

Databricks Spark Certification and Beyond

Module:
Advanced Spark

Lecture:
Broadcast Variables

Broadcast Variables in Apache Spark

Spark broadcast variables are part of the Spark low-level APIs.
If you are using Spark Dataframe APIs, you are not likely to use broadcast variables.
They were primarily used with the Spark low-level RDD APIs.
However, it is essential to understand the concept.

Let me quickly explain the requirement, and then I will explain the code. I want to create a Spark UDF and use it in my spark project. As we already know, a Spark UDF is nothing but a function that takes one or more columns as input and returns one transformed column as output. So I wanted to create UDF of the following structure.

Requirement

1. Create a UDF as following

```
my_func(product_code: str) -> str
```

2. Provide below data to UDF

```
prdCode = {"01056": "Product 1",
           "98312": "Product 2",
           "02845": "Product 3"}
```

3. UDF must convert product code to product name

This function takes a string product_code and returns a string product name.
And for translating the product code to a product name, the function references a small lookup table. Here is the lookup data structure.
I am showing a super simple and small lookup dataset here. But you can think of it as a 5 MB or 10 MB dataset in a real-life scenario.

Requirement

1. Create a UDF as following

```
my_func(product_code: str) -> str
```

2. Provide below data to UDF

```
prdCode = {"01056": "Product 1",
           "98312": "Product 2",
           "02845": "Product 3"}
```

3. UDF must convert product code to product name

We want to create a UDF and provide it with two inputs.
The first input is a product_code which goes as a function argument.
The second input is a large dataset that should be available to function as a lookup table.
This is a general requirement.
In these kinds of requirements, we want to create and use UDF.
But we also want to make available some reference data to the Spark UDF.
We cannot pass this reference data to the UDF as a function argument.
So how do we provide it to the UDF?

Requirement

1. Create a UDF as following

```
my_func(product_code: str) -> str
```

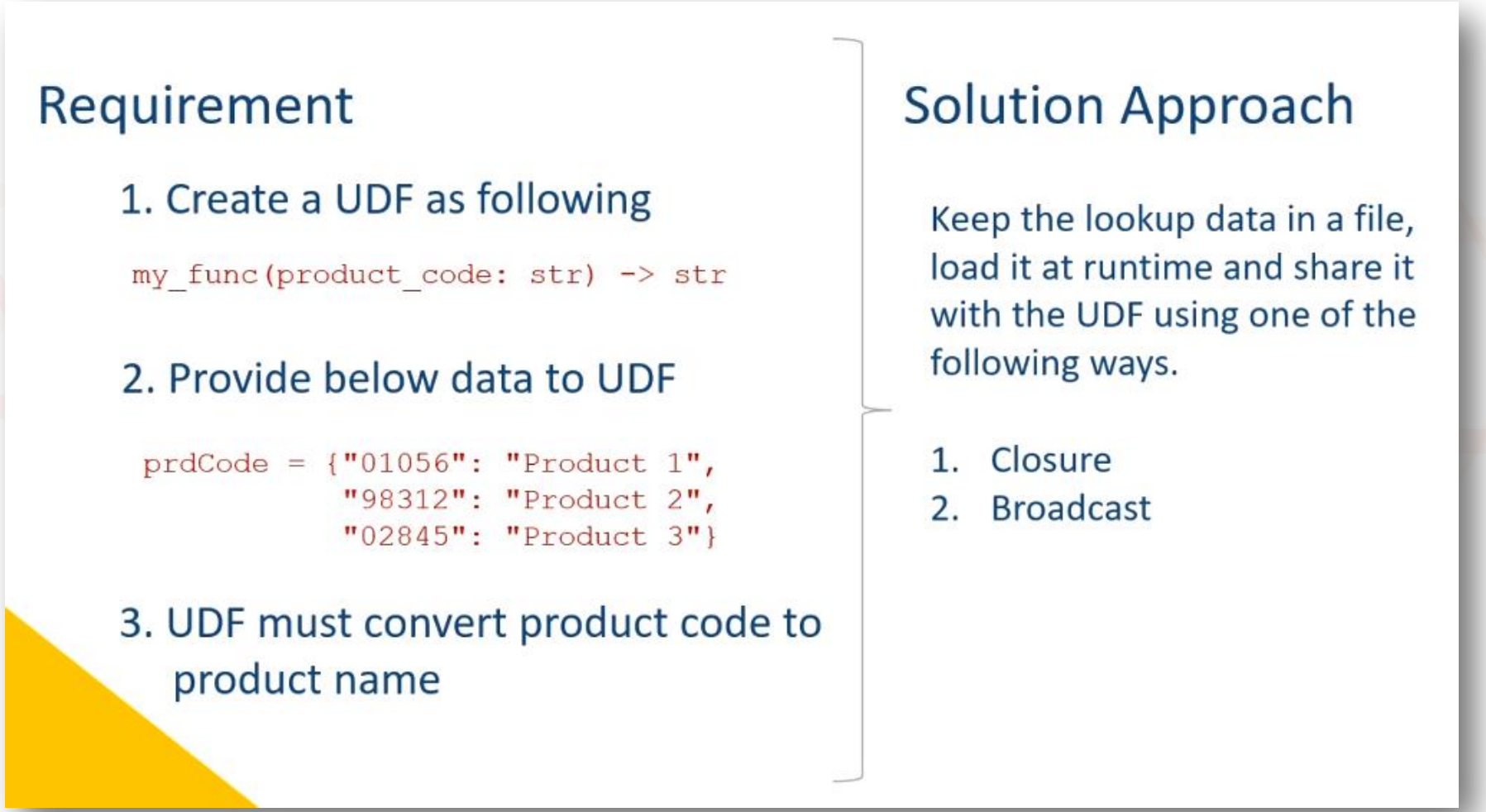
2. Provide below data to UDF

```
prdCode = {"01056": "Product 1",
           "98312": "Product 2",
           "02845": "Product 3"}
```

3. UDF must convert product code to product name

One easy way is to keep the lookup data in a file, load it at runtime and share it with the UDF.
But how do we share it with the UDF?
We have two ways to do it:

1. Closure
2. Broadcast



Here is a super simple example.
I am creating a Spark session, and you already know all this.
(Reference: https://github.com/ScholarNest/PySpark-Examples/tree/main/broadcast-for-udf)

```
6   def my_func(code: str) -> str:
7       # return prdCode.get(code)
8       return bdData.value.get(code)
9
10
11  if __name__ == "__main__":
12      spark = SparkSession \
13          .builder \
14          .appName("Demo") \
15          .master("local[3]") \
16          .getOrCreate()
17
18      prdCode = spark.read.csv("data/lookup.csv").rdd.collectAsMap()
19
20      bdData = spark.sparkContext.broadcast(prdCode)
21
22      data_list = [("98312", "2021-01-01", "1200", "01"),
23                   ("01056", "2021-01-02", "2345", "01"),
24                   ("98312", "2021-02-03", "1200", "02"),
25                   ("01056", "2021-02-04", "2345", "02"),
26                   ("02845", "2021-02-05", "9812", "02")]
27      df = spark.createDataFrame(data_list) \
28          .toDF("code", "order_date", "price", "qty")
29
30      spark.udf.register("my_udf", my_func, StringType())
31      df.withColumn("Product", expr("my_udf(code)")) \
32          .show()
```


Then I read a lookup.csv file from the storage and brought it to the Spark driver as a python dictionary.
Now the prdCode is a Python dictionary object available at the Spark driver.
And we want to share this prdCode object with my UDF.
So, I will broadcast this prdCode object to the Spark cluster.

```
6   def my_func(code: str) -> str:
7       # return prdCode.get(code)
8       return bdData.value.get(code)
9
10
11  if __name__ == "__main__":
12      spark = SparkSession \
13          .builder \
14          .appName("Demo") \
15          .master("local[3]") \
16          .getOrCreate()
17
18      prdCode = spark.read.csv("data/lookup.csv").rdd.collectAsMap()
19
20      bdData = spark.sparkContext.broadcast(prdCode)
21
22      data_list = [("98312", "2021-01-01", "1200", "01"),
23                   ("01056", "2021-01-02", "2345", "01"),
24                   ("98312", "2021-02-03", "1200", "02"),
25                   ("01056", "2021-02-04", "2345", "02"),
26                   ("02845", "2021-02-05", "9812", "02")]
27      df = spark.createDataFrame(data_list) \
28          .toDF("code", "order_date", "price", "qty")
29
30      spark.udf.register("my_udf", my_func, StringType())
31      df.withColumn("Product", expr("my_udf(code)")) \
32          .show()
```


Now, bdData is my broadcast variable. So I have two variables here. prdCode and bdData prdCode is a standard Python dictionary variable. However, the bdData is a broadcast variable.

```
6   def my_func(code: str) -> str:
7       # return prdCode.get(code)
8       return bdData.value.get(code)
9
10
11  if __name__ == "__main__":
12      spark = SparkSession \
13          .builder \
14          .appName("Demo") \
15          .master("local[3]") \
16          .getOrCreate()
17
18      prdCode = spark.read.csv("data/lookup.csv").rdd.collectAsMap()
19
20      bdData = spark.sparkContext.broadcast(prdCode)
21
22      data_list = [("98312", "2021-01-01", "1200", "01"),
23                   ("01056", "2021-01-02", "2345", "01"),
24                   ("98312", "2021-02-03", "1200", "02"),
25                   ("01056", "2021-02-04", "2345", "02"),
26                   ("02845", "2021-02-05", "9812", "02")]
27      df = spark.createDataFrame(data_list) \
28          .toDF("code", "order_date", "price", "qty")
29
30      spark.udf.register("my_udf", my_func, StringType())
31      df.withColumn("Product", expr("my_udf(code)")) \
32          .show()
```


I want to demonstrate a working example. So I am creating a Dataframe at runtime.
I will use this Dataframe to test my UDF.

```
6   def my_func(code: str) -> str:
7       # return prdCode.get(code)
8       return bdData.value.get(code)
9
10
11  if __name__ == "__main__":
12      spark = SparkSession \
13          .builder \
14          .appName("Demo") \
15          .master("local[3]") \
16          .getOrCreate()
17
18      prdCode = spark.read.csv("data/lookup.csv").rdd.collectAsMap()
19
20      bdData = spark.sparkContext.broadcast(prdCode)
21
22      data_list = [("98312", "2021-01-01", "1200", "01"),
23                   ("01056", "2021-01-02", "2345", "01"),
24                   ("98312", "2021-02-03", "1200", "02"),
25                   ("01056", "2021-02-04", "2345", "02"),
26                   ("02845", "2021-02-05", "9812", "02")]
27      df = spark.createDataFrame(data_list) \
28          .toDF("code", "order_date", "price", "qty")
29
30      spark.udf.register("my_udf", my_func, StringType())
31      df.withColumn("Product", expr("my_udf(code)")) \
32          .show()
```


In the following two lines, I am registering my UDF function as Spark SQL UDF and applying it to the Dataframe. All of this is a super simple and standard method for using a Spark UDF.

```
6   def my_func(code: str) -> str:
7       # return prdCode.get(code)
8       return bdData.value.get(code)
9
10
11  if __name__ == "__main__":
12      spark = SparkSession \
13          .builder \
14          .appName("Demo") \
15          .master("local[3]") \
16          .getOrCreate()
17
18      prdCode = spark.read.csv("data/lookup.csv").rdd.collectAsMap()
19
20      bdData = spark.sparkContext.broadcast(prdCode)
21
22      data_list = [("98312", "2021-01-01", "1200", "01"),
23                   ("01056", "2021-01-02", "2345", "01"),
24                   ("98312", "2021-02-03", "1200", "02"),
25                   ("01056", "2021-02-04", "2345", "02"),
26                   ("02845", "2021-02-05", "9812", "02")]
27      df = spark.createDataFrame(data_list) \
28          .toDF("code", "order_date", "price", "qty")
29
30      spark.udf.register("my_udf", my_func, StringType())
31      df.withColumn("Product", expr("my_udf(code)")) \
32          .show()
```


Now, let's scroll up to the UDF definition.
I am accepting the product code as input and returning the product name from the bdData broadcast variable.
So what is happening here?
My UDF function will be called for each record in my Dataframe.
While calling the UDF, I am passing the product_code column to the UDF.
The UDF will look for the product_code in the lookup and return the corresponding product name.

```
6   def my_func(code: str) -> str:
7       # return prdCode.get(code)
8       return bdData.value.get(code)
9
10
11  if __name__ == "__main__":
12      spark = SparkSession \
13          .builder \
14          .appName("Demo") \
15          .master("local[3]") \
16          .getOrCreate()
17
18      prdCode = spark.read.csv("data/lookup.csv").rdd.collectAsMap()
19
20      bdData = spark.sparkContext.broadcast(prdCode)
21
22      data_list = [("98312", "2021-01-01", "1200", "01"),
23                   ("01056", "2021-01-02", "2345", "01"),
24                   ("98312", "2021-02-03", "1200", "02"),
25                   ("01056", "2021-02-04", "2345", "02"),
26                   ("02845", "2021-02-05", "9812", "02")]
27      df = spark.createDataFrame(data_list) \
28          .toDF("code", "order_date", "price", "qty")
29
30      spark.udf.register("my_udf", my_func, StringType())
31      df.withColumn("Product", expr("my_udf(code)")) \
32          .show()
```


I am using the broadcast variable here.
But You can also use the prdCode directly.
Both the approaches are going to work.
But if you are using prdCode, you are using a lambda closure.
And if you are using bdData, you are using a broadcast variable.

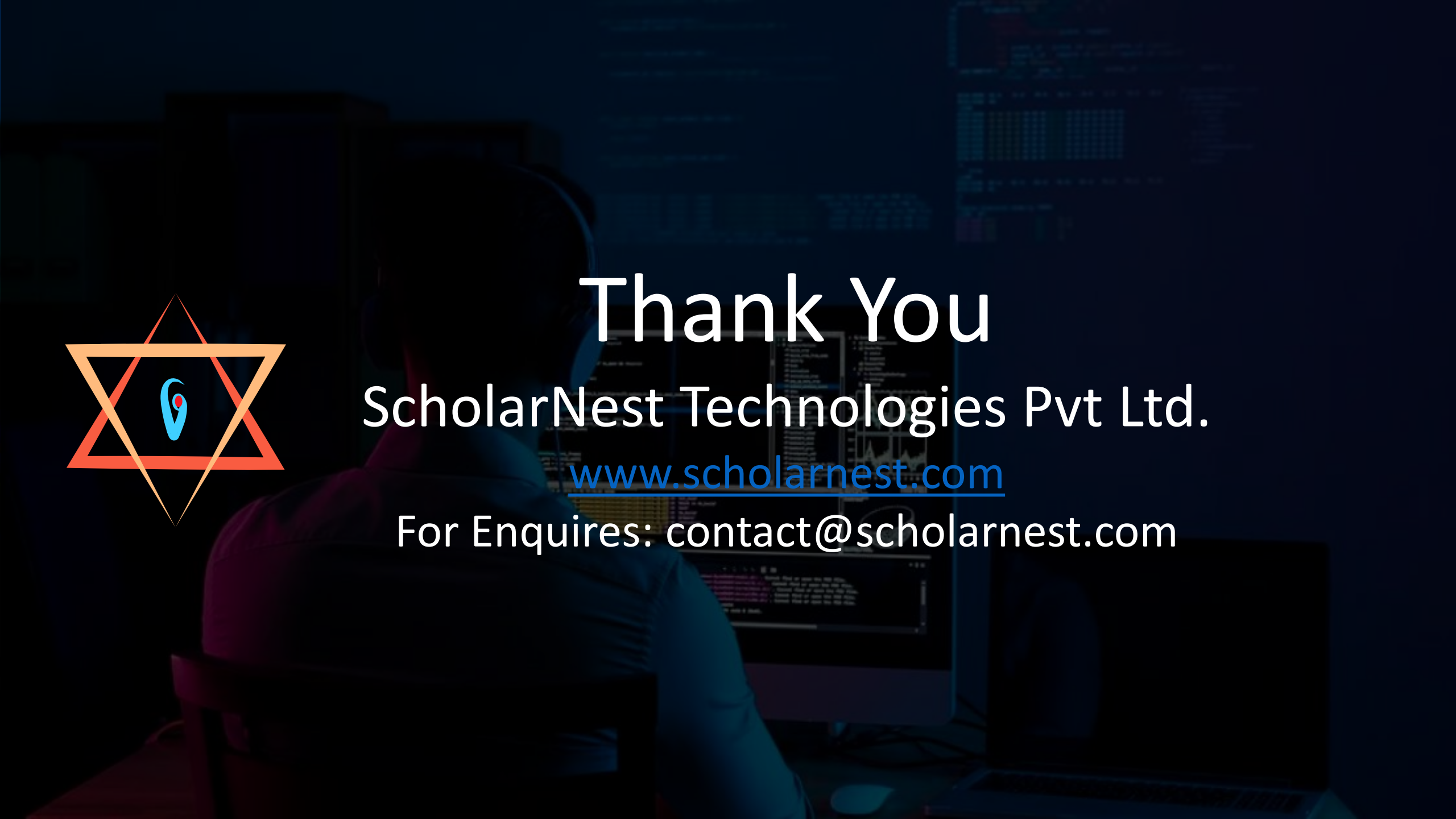
```
6   def my_func(code: str) -> str:
7       # return prdCode.get(code)
8       return bdData.value.get(code)
9
10
11  if __name__ == "__main__":
12      spark = SparkSession \
13          .builder \
14          .appName("Demo") \
15          .master("local[3]") \
16          .getOrCreate()
17
18      prdCode = spark.read.csv("data/lookup.csv").rdd.collectAsMap()
19
20      bdData = spark.sparkContext.broadcast(prdCode)
21
22      data_list = [("98312", "2021-01-01", "1200", "01"),
23                   ("01056", "2021-01-02", "2345", "01"),
24                   ("98312", "2021-02-03", "1200", "02"),
25                   ("01056", "2021-02-04", "2345", "02"),
26                   ("02845", "2021-02-05", "9812", "02")]
27      df = spark.createDataFrame(data_list) \
28          .toDF("code", "order_date", "price", "qty")
29
30      spark.udf.register("my_udf", my_func, StringType())
31      df.withColumn("Product", expr("my_udf(code)")) \
32          .show()
```


But what is the difference between using the broadcast variable and using the prdCode directly?
If you are using a closure, Spark will serialize the closure to each task.
If you have 1000 tasks running on a 30 node cluster, your closure is serialized 1000 times.

But if you are using a broadcast variable, Spark will serialize it once per worker node and cache it there for future usage.
So you have 1000 tasks and 30 worker nodes.
In that case, the broadcast variable is serialized only 30 times, once per worker node.

Let us summarize.
Broadcast variables are shared, immutable variables cached on every machine in the cluster instead of serialized with every single task. And this serialization is lazy. What does it mean?
The broadcast variable is serialized to a worker node only if you have at least one task running on the worker node that needs to access the broadcast variable.
However, you should also remember that the broadcast variable must fit into the memory of an executor.

One last thing before we close this lecture.
Spark Dataframe APIs use this same technique to implement broadcast joins.
So they will bring a small table to the driver as we did in this example.
Then they will broadcast it to the workers for being used in the join.
So if you carefully design your application logic, you can meet your requirement using broadcast joins in most cases.
However, if you plan to use UDF and make some datasets available to your UDF for reference, you can use the broadcast variables, as I demonstrated in our example.



Thank You

ScholarNest Technologies Pvt Ltd.

www.scholarnest.com

For Enquires: contact@scholarnest.com

Spark Azure Databricks

Databricks Spark Certification and Beyond

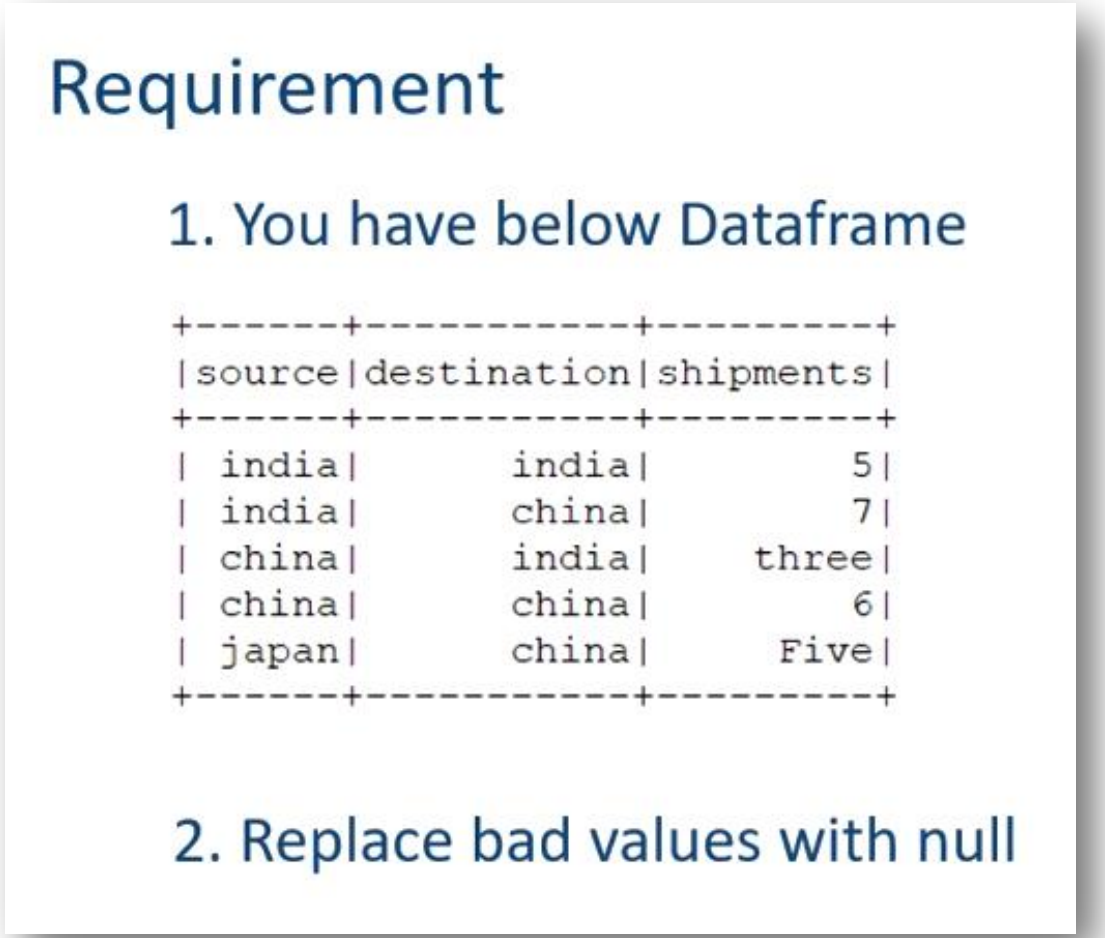
Module:
Advanced Spark

Lecture:
Spark Accumulators

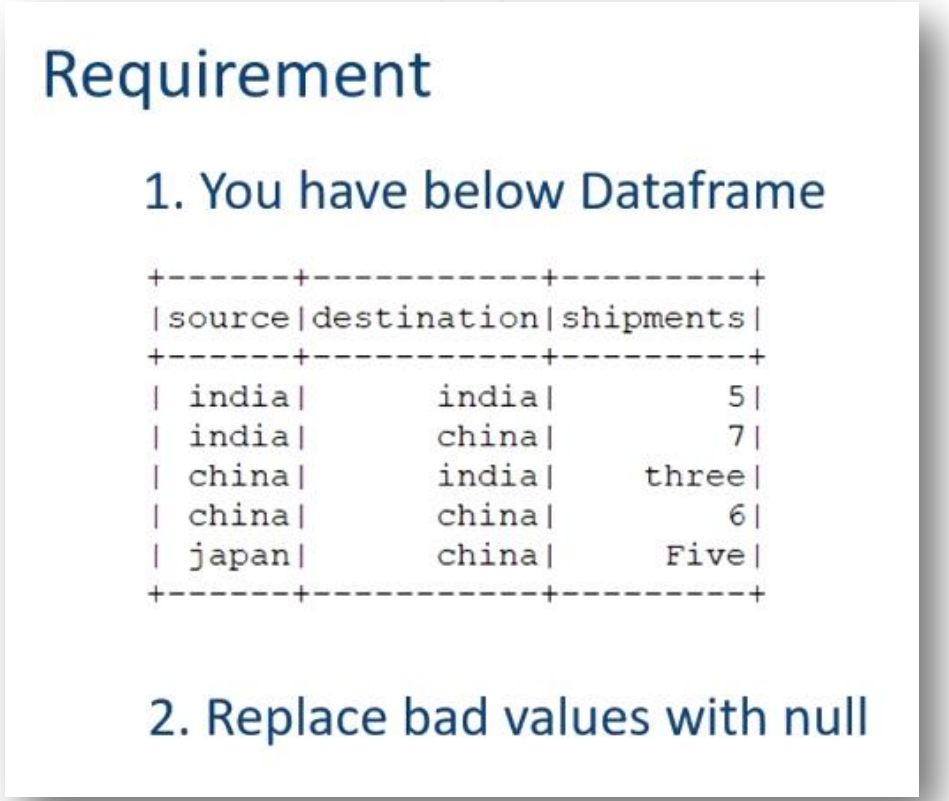
Spark Accumulators

Spark accumulators are part of the Spark low-level APIs.
If you are using Spark Dataframe APIs, you are not likely to use accumulators.
They were primarily used with the Spark low-level RDD APIs.
However, it is essential to understand the concept.
So let me explain it at a high level.
We will use an example code to explain the concept.

Let me quickly explain the requirement, and then I will explain the code. I want to create a Spark UDF and use it in my spark project. As we already know, a Spark UDF is nothing but a function that takes one or more columns as input and returns one transformed column as output. So I wanted to create UDF of the following structure.



This Dataframe represents an aggregated shipment record. We have a source column, then a destination column, and finally, we have a shipments column. The shipments column represents the number of shipments from the given source to a destination. However, we have a slight problem here. The shipment column is expected to be an integer column.
But we have some bad records in this table. I want to fix these bad records. But how do we do it? We are asked to take null if the shipments count is not a valid integer.



So I decided to create a UDF for this. Here is the code for the UDF shown below.
This UDF simply takes the shipments column, converts it into an integer, and returns it.
If we cannot convert the shipments column to an integer, we return null.

Requirement

1. You have below Dataframe

```
+------+-----------+---------+
|source|destination|shipments|
+------+-----------+---------+
| india|      india|        5|
| india|      china|        7|
| china|      india|    three|
| china|      china|        6|
| japan|      china|     Five|
+------+-----------+---------+
```

2. Replace bad values with null

Create a UDF to handle bad records

```
6   def handle_bad_rec(shipments: str) -> int:
7       s = None
8       try:
9           s = int(shipments)
10      except ValueError:
11          None
12      return s
```

How to use this UDF?
Here is the code given below.
So I will register the UDF and use it to create a new column with the correct values.

Use the UDF to fix bad records

```
32      spark.udf.register("udf_handle_bad_rec", handle_bad_rec, IntegerType())
33      df.withColumn("shipments_int", expr("udf_handle_bad_rec(shipments)")) \
34          .show()
```


Here is the expected output.
I replaced the bad values with nulls.

Expected Output

source	destination	shipments	shipments_int
india	india	5	5
india	china	7	7
china	india	three	null
china	china	6	6
japan	china	Five	null

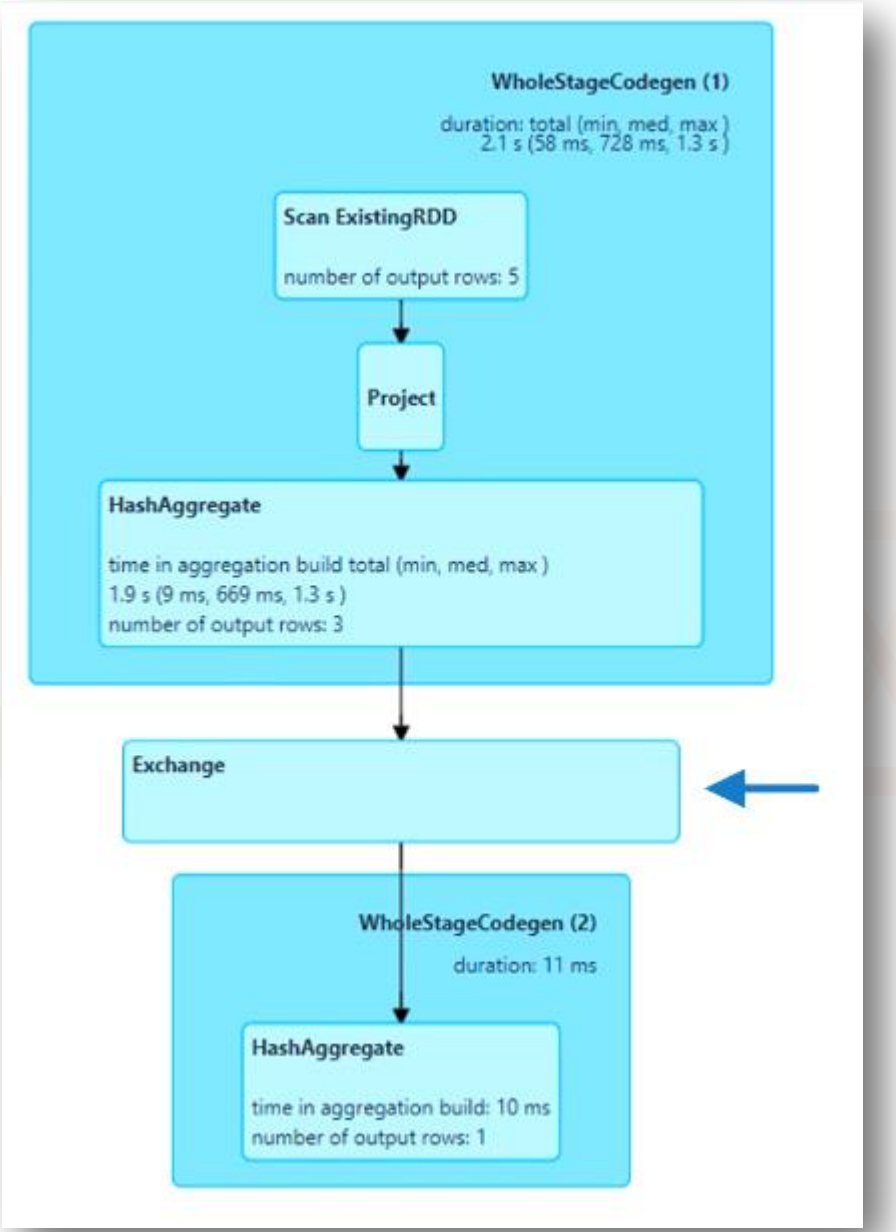
But now, I have another requirement.
We also want to count the number of bad records.
How to do it?
Can you think of some solution? We have an easy way to do it. I can simply count the nulls in the shipments_int column. We fixed two rows in this Dataframe, replacing two bad values with nulls. So If I count the nulls in the new column, I exactly know the number of bad records. The solution is perfectly fine.

Count Nulls

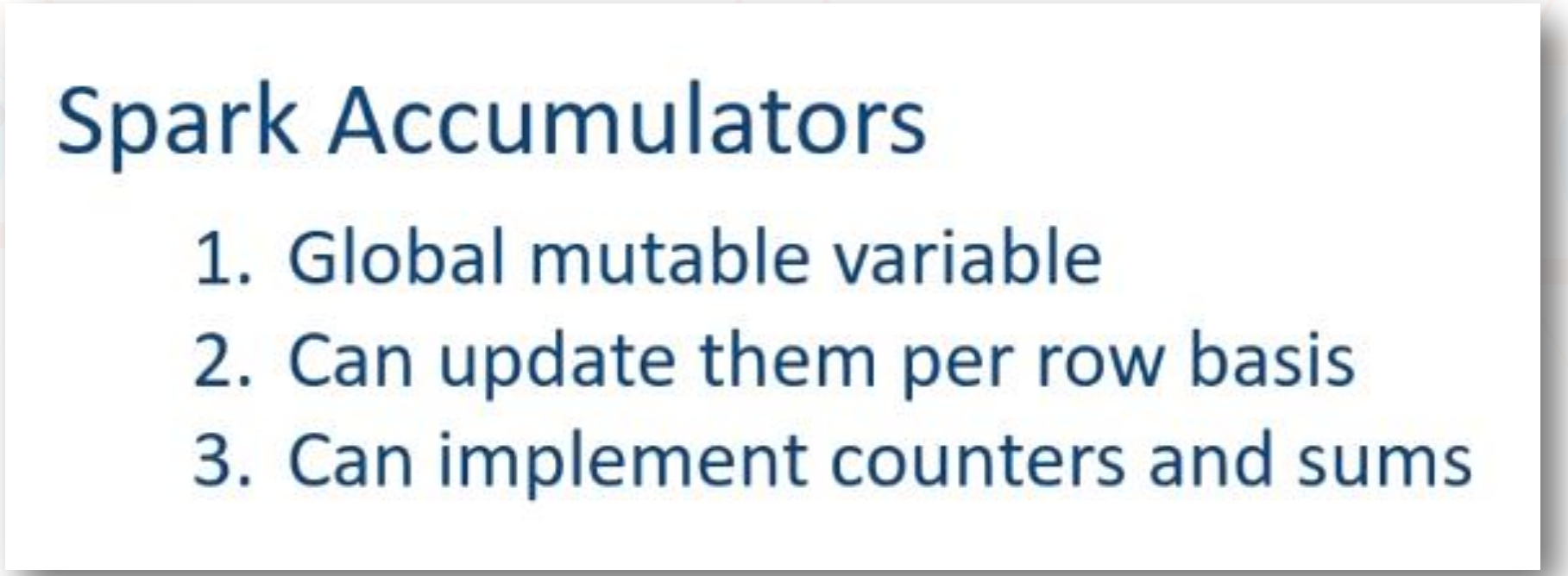
source	destination	shipments	shipments_int
india	india	5	5
india	china	7	7
china	india	three	null
china	china	6	6
japan	china	Five	null

But count() aggregation or a count() action on a Dataframe has a wide dependency.
Spark will add one extra stage and a shuffle operation.
My execution plan looks like this.

So you can see this exchange in my execution plan.
This exchange represents a shuffle that is caused by the count() operation.
And that's not a good thing. I don't like this shuffle exchange in my plan?
Can I do something else and avoid this shuffle?



I wish I had a global variable to increment it from my UDF while I am fixing the bad record.
And that's precisely what Spark Accumulators are.
Spark Accumulator is a global mutable variable that a Spark cluster can safely update on a per-row basis.
You can use them to implement counters or sums.



Here is the complete example code.
I created an accumulator using the spark context with an initial value of zero. So the bad_rec is an accumulator variable.

```
6   def handle_bad_rec(shipments: str) -> int:
7       s = None
8       try:
9           s = int(shipments)
10      except ValueError:
11          bad_rec.add(1)
12      return s
13
```

```
15  if __name__ == "__main__":
16      spark = SparkSession \
17          .builder \
18          .appName("Demo") \
19          .master("local[3]") \
20          .getOrCreate()
21
22      data_list = [("india", "india", '5'),
23                   ("india", "china", '7'),
24                   ("china", "india", 'three'),
25                   ("china", "china", '6'),
26                   ("japan", "china", 'Five')]
27
28      df = spark.createDataFrame(data_list) \
29          .toDF("source", "destination", "shipments")
30
31      bad_rec = spark.sparkContext.accumulator(0)
32      spark.udf.register("udf_handle_bad_rec", handle_bad_rec, IntegerType())
33      df.withColumn("shipments_int", expr("udf_handle_bad_rec(shipments)")) \
34          .show()
35
36      print("Bad Record Count:" + str(bad_rec.value))
37
```


Then I use this accumulator variable in my UDF. So whenever I see a bad record, I will increment the accumulator by one.

```
6   def handle_bad_rec(shipments: str) -> int:
7       s = None
8       try:
9           s = int(shipments)
10      except ValueError:
11          bad_rec.add(1)
12      return s
13
```

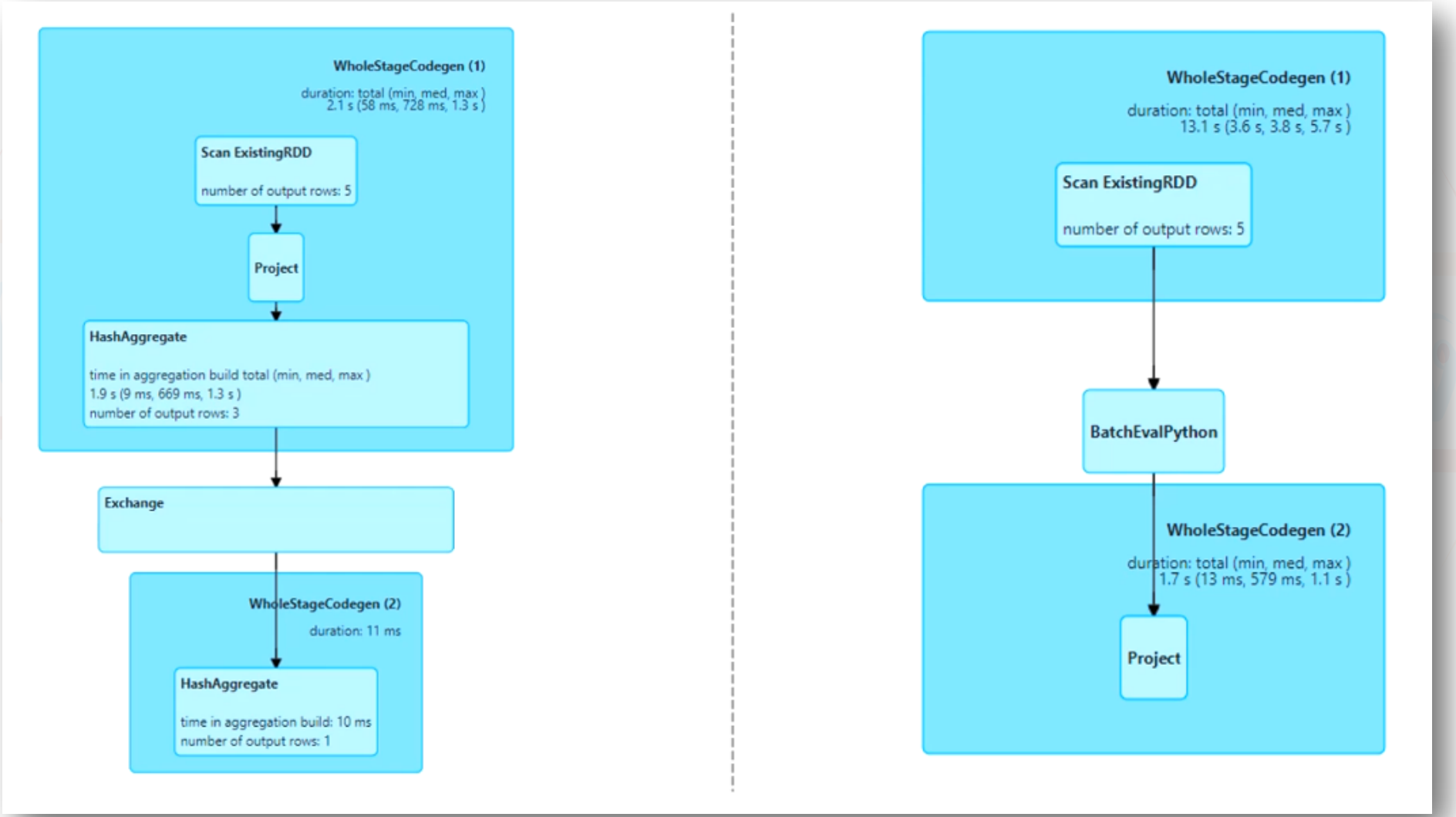
```
15  if __name__ == "__main__":
16      spark = SparkSession \
17          .builder \
18          .appName("Demo") \
19          .master("local[3]") \
20          .getOrCreate()
21
22      data_list = [("india", "india", '5'),
23                   ("india", "china", '7'),
24                   ("china", "india", 'three'),
25                   ("china", "china", '6'),
26                   ("japan", "china", 'Five')]
27
28      df = spark.createDataFrame(data_list) \
29          .toDF("source", "destination", "shipments")
30
31      bad_rec = spark.sparkContext.accumulator(0)
32      spark.udf.register("udf_handle_bad_rec", handle_bad_rec, IntegerType())
33      df.withColumn("shipments_int", expr("udf_handle_bad_rec(shipments)")) \
34          .show()
35
36      print("Bad Record Count:" + str(bad_rec.value))
37
```


Now come back to the end of the program.
This accumulator variable is maintained at the driver. So I can simply print the final value.
I do not have to collect it from anywhere. Because the accumulators always live at the driver.
All the tasks will increment the value of this accumulator using internal communication with the driver.

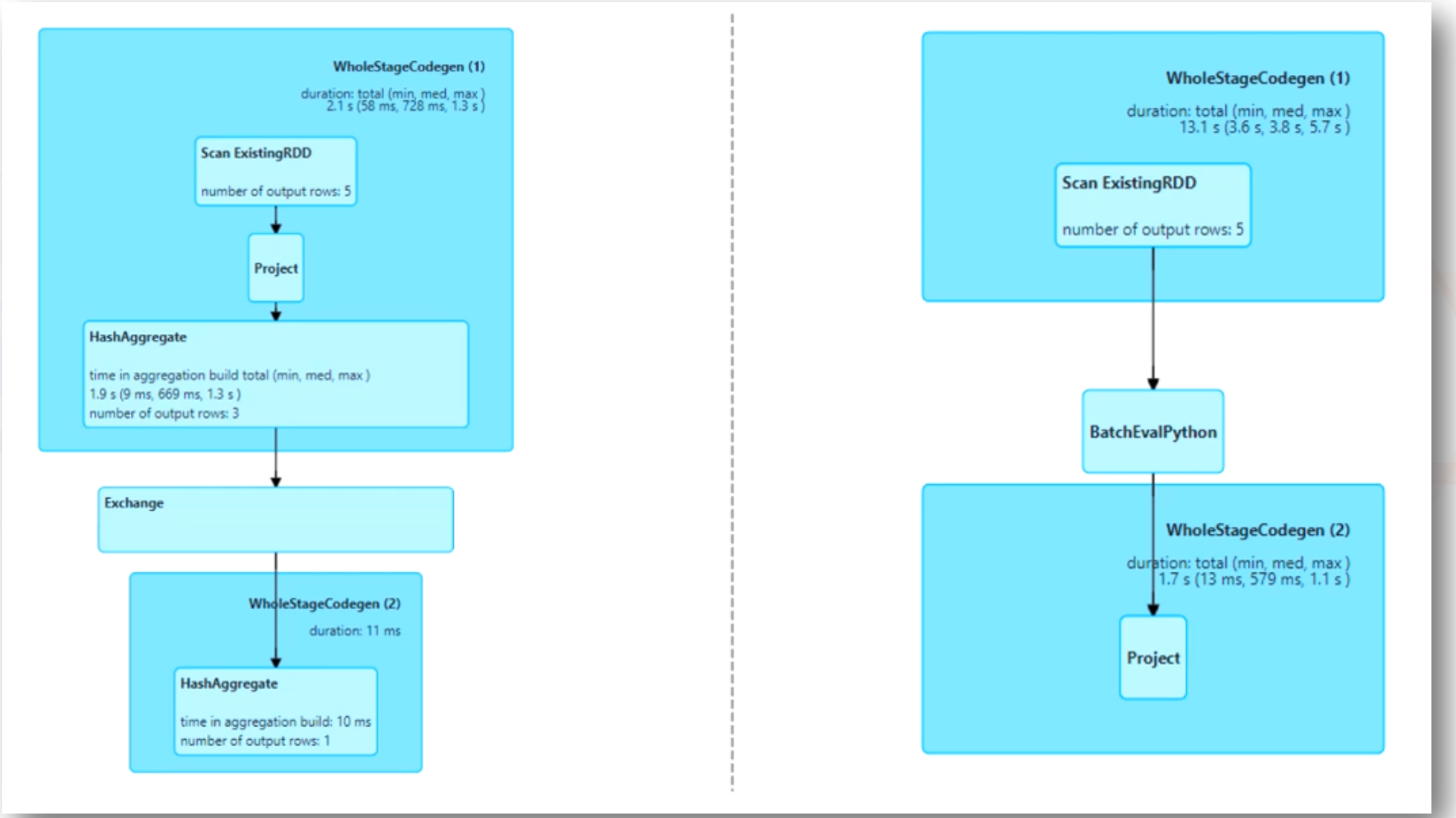
```
6   def handle_bad_rec(shipments: str) -> int:
7       s = None
8       try:
9           s = int(shipments)
10      except ValueError:
11          bad_rec.add(1)
12      return s
13
```

```
15  if __name__ == "__main__":
16      spark = SparkSession \
17          .builder \
18          .appName("Demo") \
19          .master("local[3]") \
20          .getOrCreate()
21
22      data_list = [("india", "india", '5'),
23                   ("india", "china", '7'),
24                   ("china", "india", 'three'),
25                   ("china", "china", '6'),
26                   ("japan", "china", 'Five')]
27
28      df = spark.createDataFrame(data_list) \
29          .toDF("source", "destination", "shipments")
30
31      bad_rec = spark.sparkContext.accumulator(0)
32      spark.udf.register("udf_handle_bad_rec", handle_bad_rec, IntegerType())
33      df.withColumn("shipments_int", expr("udf_handle_bad_rec(shipments)")) \
34          .show()
35
36      print("Bad Record Count:" + str(bad_rec.value))
37
```

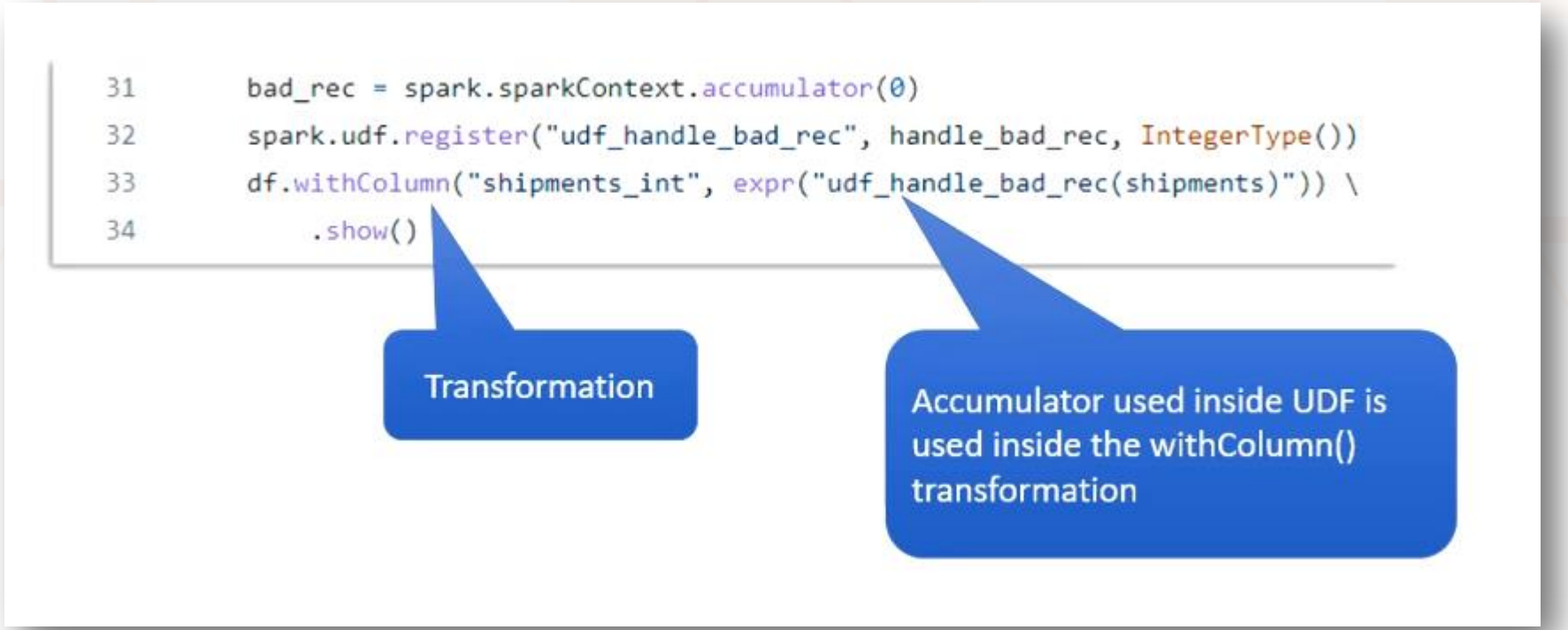

Here is the execution plan.
The first is the old plan with a count(), and the second uses an accumulator variable. I saved a shuffle and a hash aggregation. So that's one potential use of an accumulator.



You can use accumulators for counting whatever you want to count while processing your data. They are similar to global counter variables in spark.



I used the Spark accumulator from inside a withColumn() transformation. I mean, I used it from inside the UDF, but I am calling the UDF inside the withColumn() transformation.
So effectively, the accumulator is used inside the withColumn() transformation. That's perfectly fine. But some people use the accumulator from inside an action. For example, we also have a forEach() method on a PySpark Dataframe. That's an action. The forEach() action takes a lambda and applies the lambda function on each row. You can also use an accumulator from within the forEach() action.

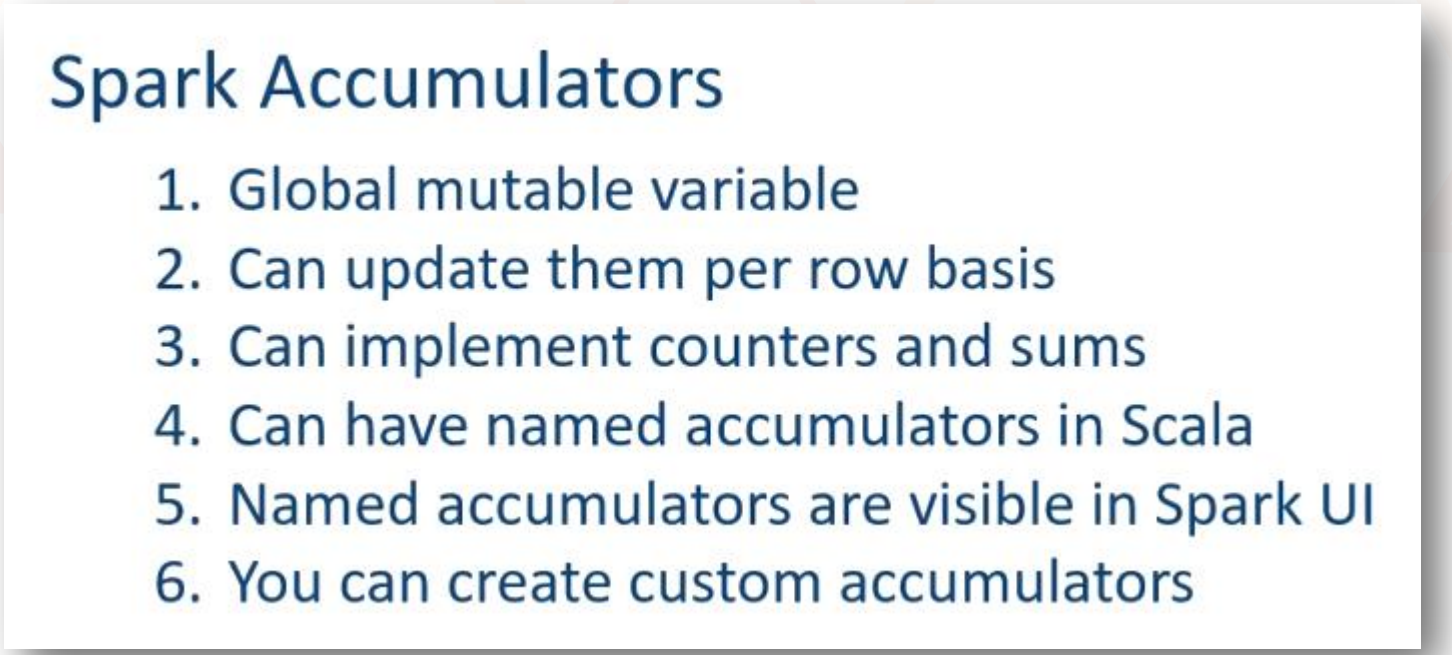


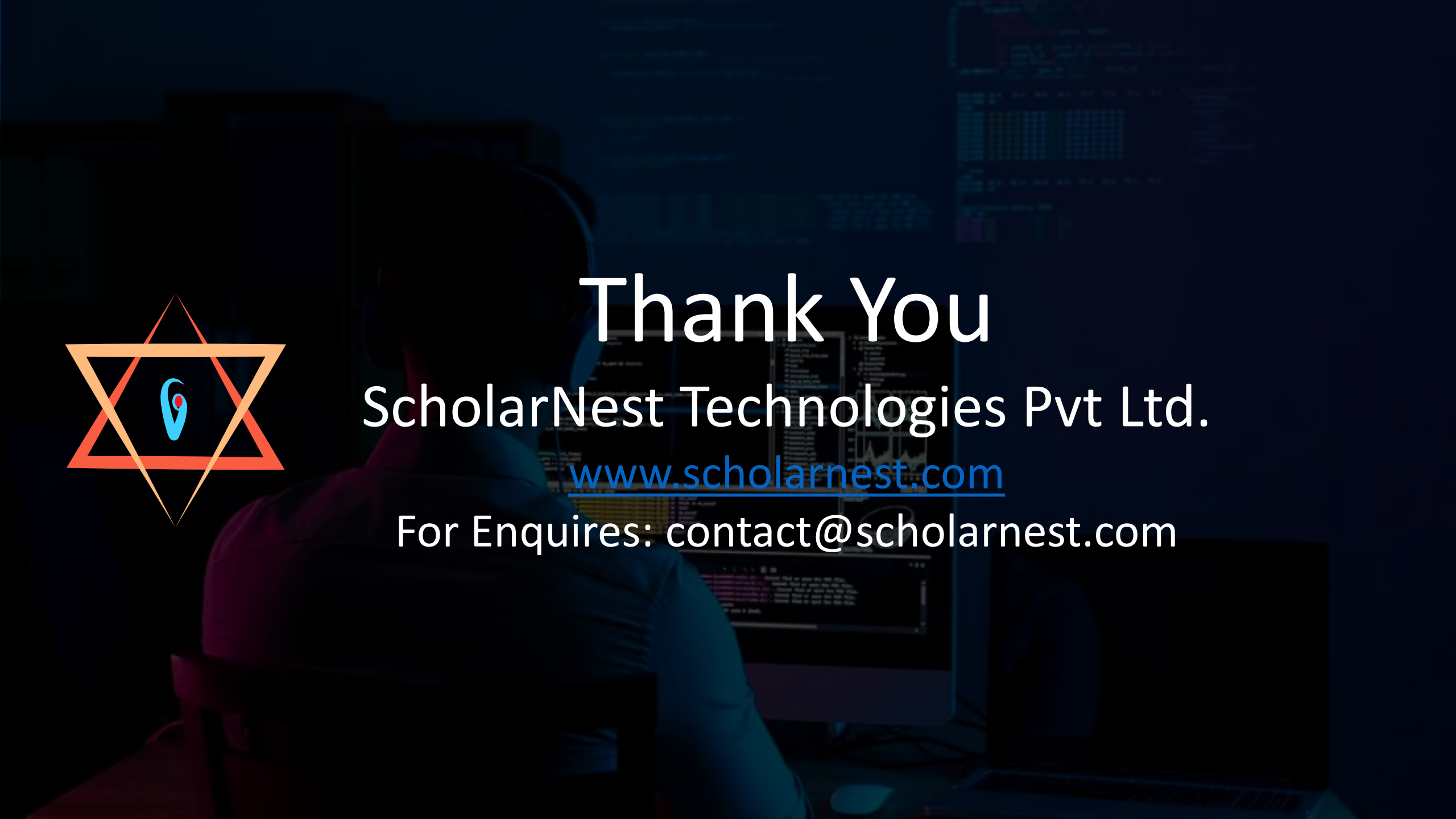
You can increment your accumulators from inside a transformation method or from inside an action method.
But it is always recommended to use an accumulator from inside action and avoid using it from inside a transformation.
Why? Because Spark gives you a guarantee of accurate results when the accumulator is incremented from inside an action.
We already know, some spark tasks can fail for a variety of reasons.
But the driver will retry those tasks on a different worker assuming success in the retry.
Spark may also trigger a duplicate task if a task is running very slow.

Spark runs duplicate tasks in many situations.
So if a task is running 2-3 times on the same data, it will increment your accumulator multiple times, distorting your counter.
But if you are incrementing your accumulator from inside an action, Spark guarantees accurate results.
So Spark guarantees that each task's update to the accumulator will be applied only once, even if the task is restarted.
But if you are incrementing your accumulator from inside a transformation, Spark doesn't give this guarantee.

That's all about accumulators except two last points:

1. Spark in Scala also allows you to give your accumulator a name and show them in the Spark UI. However, PySpark accumulators are always unnamed, and they do not show up in the Spark UI.
2. Spark allows you to create Long and Float accumulators. However, you can also create some custom accumulators. But that part is outside the scope of this course because accumulators are now used only for a few specific scenarios.





Thank You

ScholarNest Technologies Pvt Ltd.

www.scholarnest.com

For Enquires: contact@scholarnest.com

Spark Azure Databricks

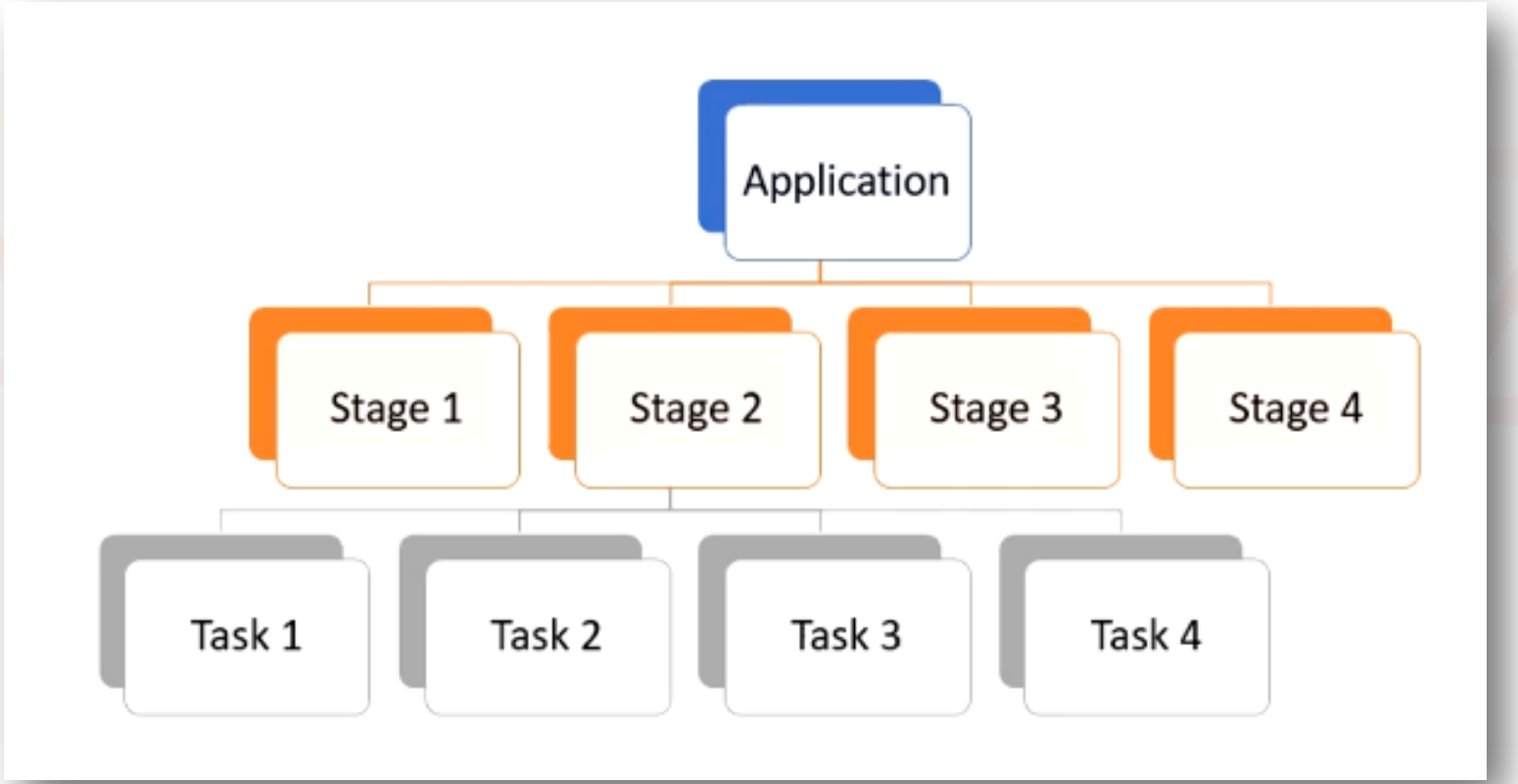
Databricks Spark Certification and Beyond

Module:
Advanced Spark

Lecture:
Spark Speculative Execution

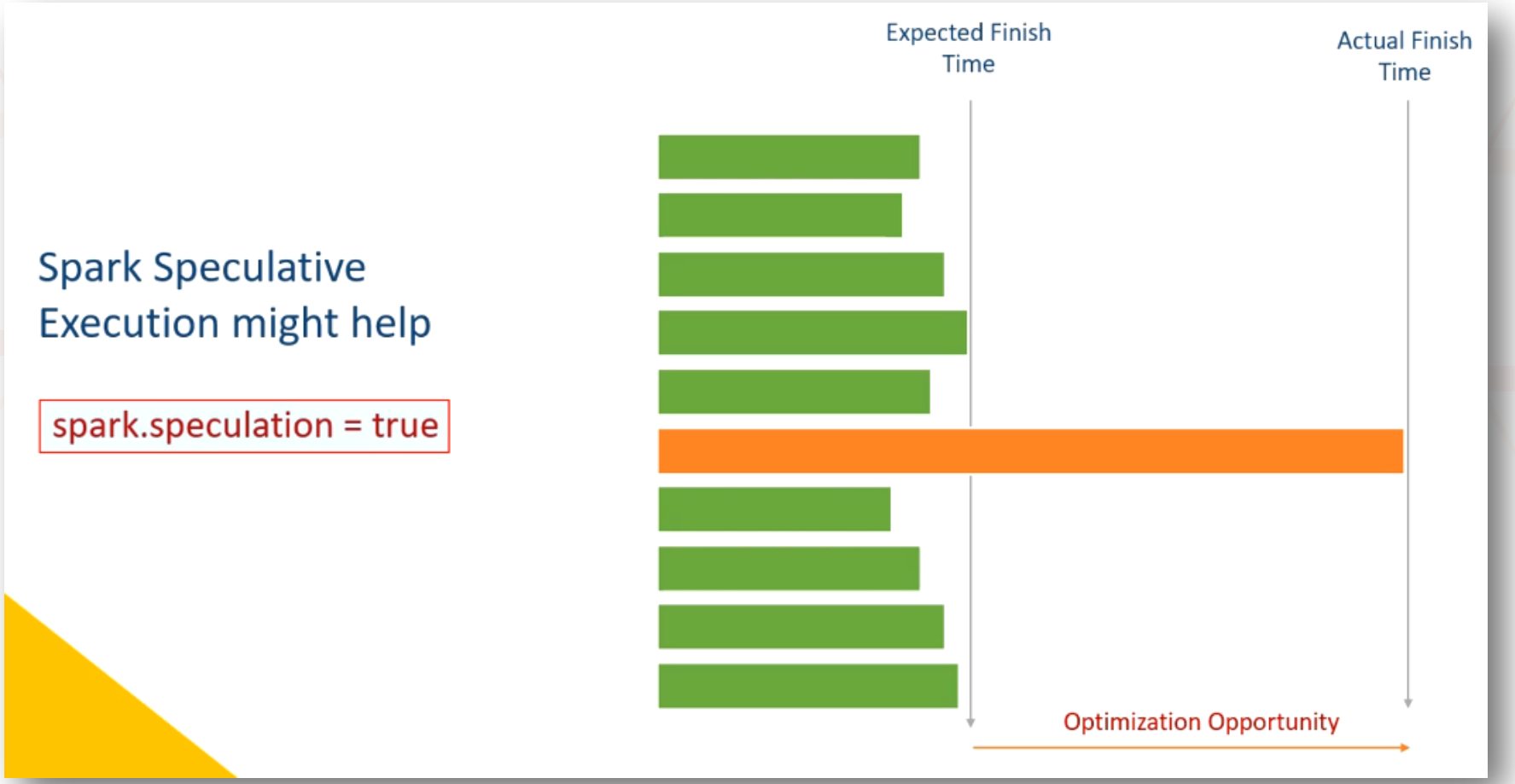
Spark Speculative Execution

We already know that Apache Spark runs your application as a set of jobs.
These jobs are broken down into stages, and each stage is executed in parallel tasks.
So a task is the smallest unit of work in Apache Spark.



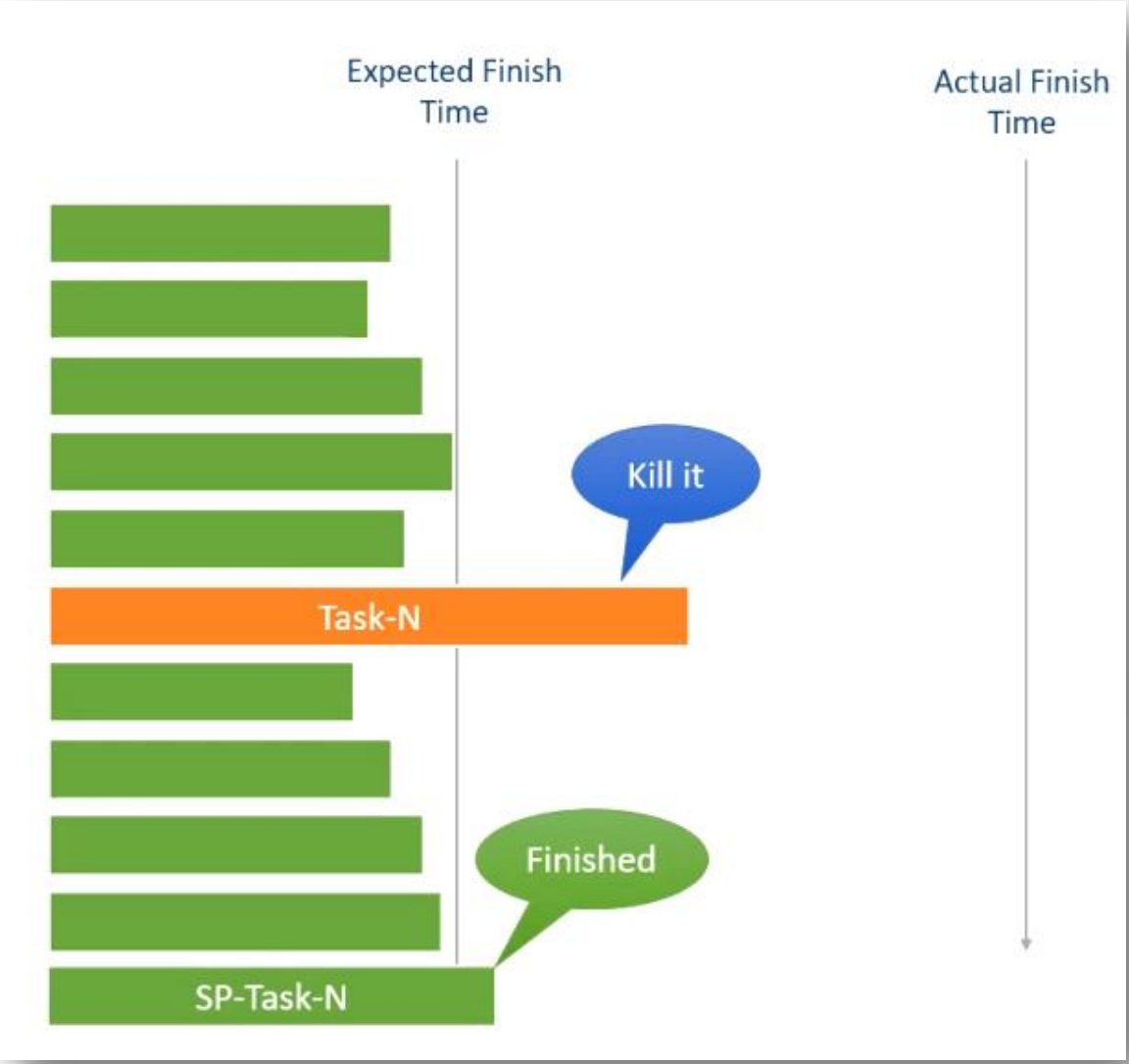
Now let's assume your Spark job is running ten tasks.
You looked at the Spark UI and noticed that out of those ten tasks. All other tasks take almost equal time to complete except one task.
One task takes a much longer time to complete.
However, a Spark job stage is not complete until all the tasks of that stage are complete.
So looking at this diagram, you realized that one task is delaying your stage.
If this task is also completed along with other tasks, your stage will complete faster, and you can reduce the Spark job execution time.

So what can you do? One easy thing is to enable Spark Speculative execution. You can enable it using the following configuration highlighted below. By default, Spark speculative execution is set to false. You can set it to true and enable it. Once you enable speculative execution, Spark will automatically identify the slow-running tasks and run a duplicate copy of the same tasks on other nodes.



The idea is to start a copy of the same task on another worker node and see if that runs faster. This new duplicate task is known as a speculative task. Now you have two copies of the task. So Spark will accept the task that finishes first and kill the other task. These speculative tasks can solve your problem in some cases. However, speculative tasks are not silver bullets, and they do not solve every slow-running task problem. They are helpful if and only if the task is slow due to a problem with the worker node.

Your task may be running on a faulty worker node, performing slowly due to some hardware problem, or struggling with overload.
It could be anything specific to the worker node that is causing your task to run slow.
If we restart that task on a different worker node, it might run faster.
And that's the main idea behind the Spark speculative task execution.
Spark will automatically identify a slow-running task, run a duplicate copy of the same on a different worker node, and see if that runs faster.
Finally, Spark will take only one faster task from these two and kill the other one.



However, you should also remember that enabling speculative execution will start taking some extra resources from your Spark cluster.
Running speculative tasks is an overhead, and in many cases, it might not give you any benefits. That's why it is disabled by default. A speculative task cannot help you in the following situation:

1. If your task is running slow due to a data skew problem
2. It is slow because the task is struggling with low memory.

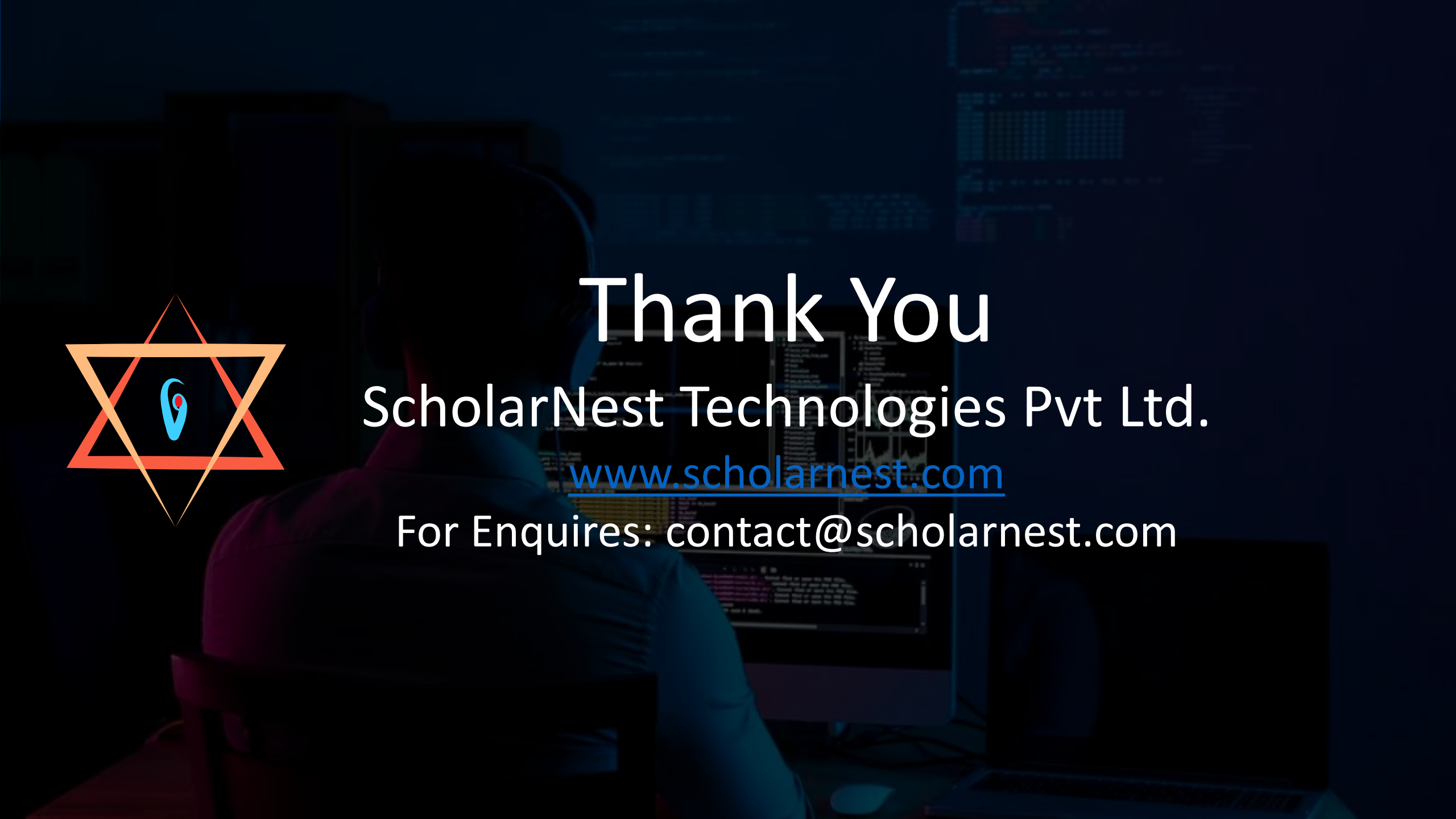
Spark doesn't know the root cause of the slow-running task, and it simply starts a speculative task to see if that one finishes faster.
So be careful before enabling speculative execution.

If you have some extra resources, enabling speculation might be a good thing to do. Spark also offers you some additional configurations to fine-tune the speculation as follows:

1. ***spark.speculation.interval = 100ms*** : The default value of the speculation interval is 100 ms. So Spark will check for task slowness and see if a speculative task is needed. And Spark will perform that check in a 100 ms loop.
2. ***spark.speculation.multiplier = 1.5*** : The default value of the speculation multiplier is 1.5. So a task is speculative only if it takes more than 1.5 times the median of other tasks' execution time. So if the median value of other tasks is 5 seconds, then a task can be speculative if and only if it takes more than 7.5 seconds.
3. ***spark.speculation.quantile = 0.75*** : The third configuration represents a fraction of tasks that must complete before speculation is enabled for a particular stage. The default value is 75%. So if more than 75% of tasks are complete and one task is still struggling, Spark will consider starting a speculative task.

4. ***spark.speculation.minTaskRuntime = 100ms*** : The next one is the minimum amount of time a task runs before being considered for speculation. This can be used to avoid launching speculative copies of tasks that are very short.
5. ***spark.speculation.task.duration.threshold = None*** : The last one is a hard limit of task duration, after which the scheduler would try to speculative run the task.

So Spark understands that the speculation consumes extra resources, it is a costly solution, and it may not solve the problem of slow-running tasks.
And hence, they offer you a bunch of configurations to fine-tune the speculation so you can protect yourself from running unnecessary speculative tasks.



Thank You

ScholarNest Technologies Pvt Ltd.

www.scholarnest.com

For Enquires: contact@scholarnest.com

Spark Azure Databricks

Databricks Spark Certification and Beyond

Module:
Advanced Spark

Lecture:
Dynamic Resource Allocation

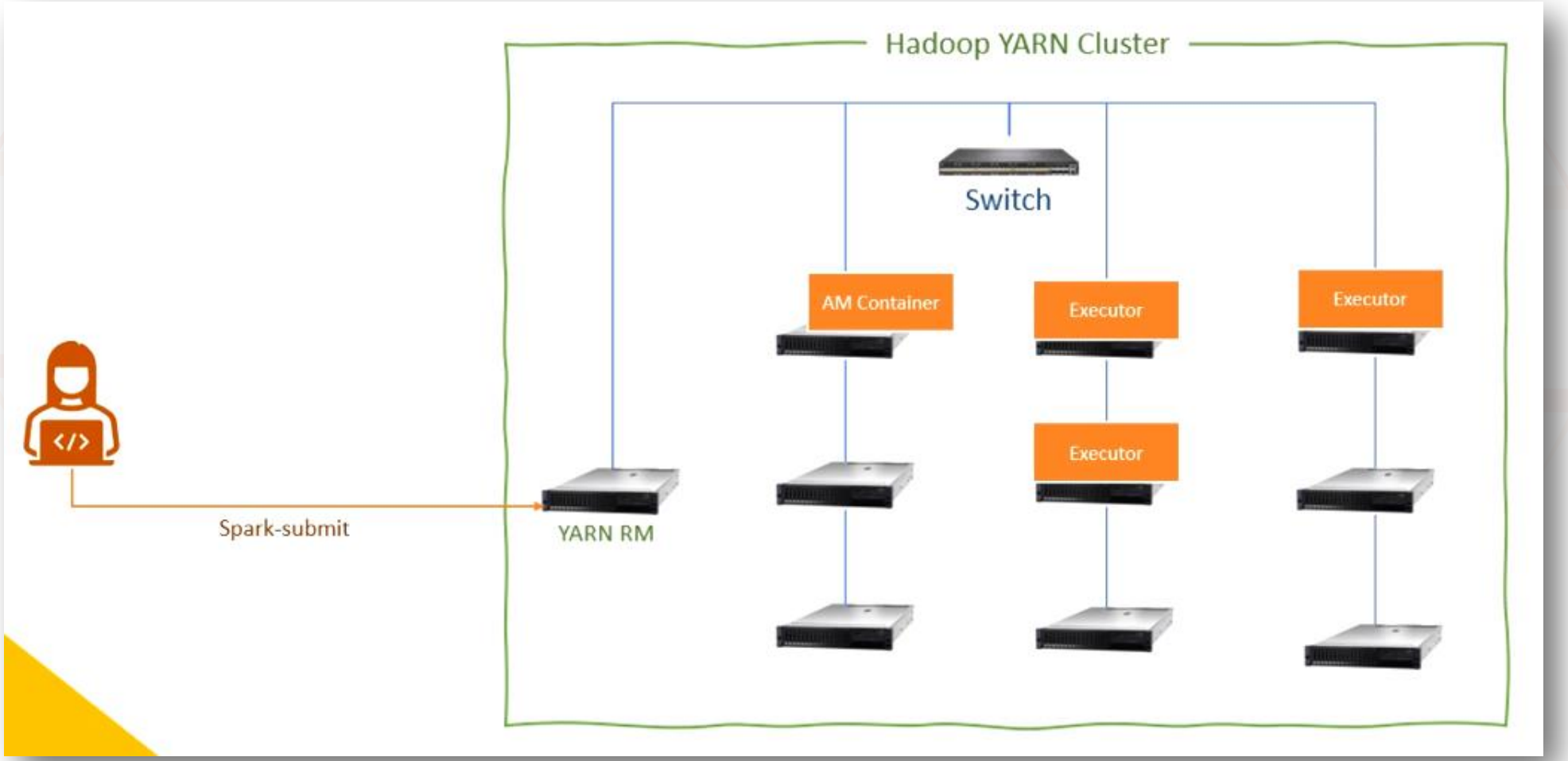
Dynamic Resource Allocation in Spark

When we talk about scheduling in the context of Apache Spark, we mean two things.

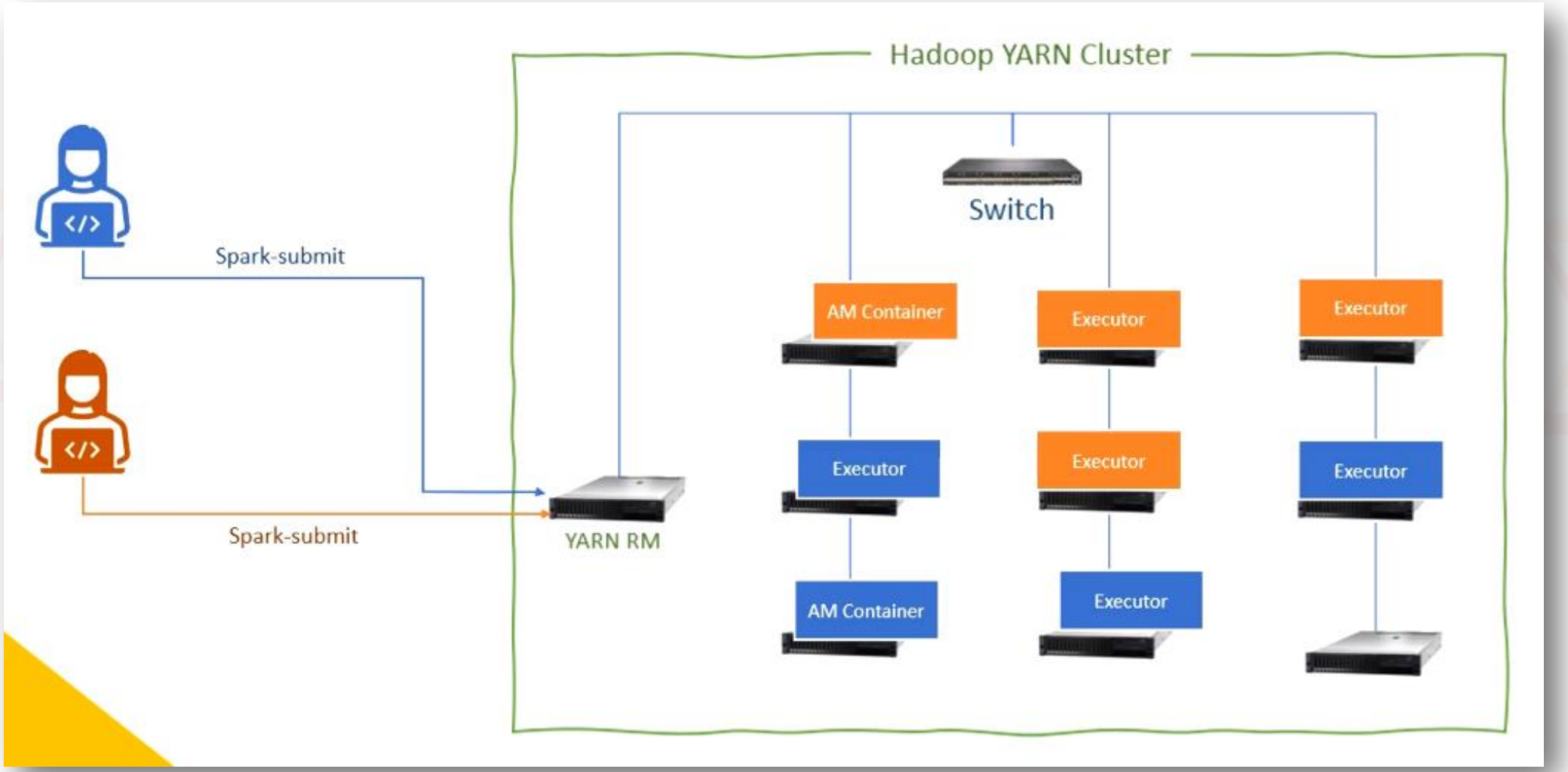
1. Scheduling Across Applications
2. Scheduling Within an Application

So let's try to understand the first scenario of scheduling across applications.

Spark is a distributed application, and it runs on a cluster. So we submit an application to a cluster. The cluster manager allocates some resources, and your application starts on the cluster. What if we submit two applications to the same cluster. Or what if some other user also submits an application to the same cluster.

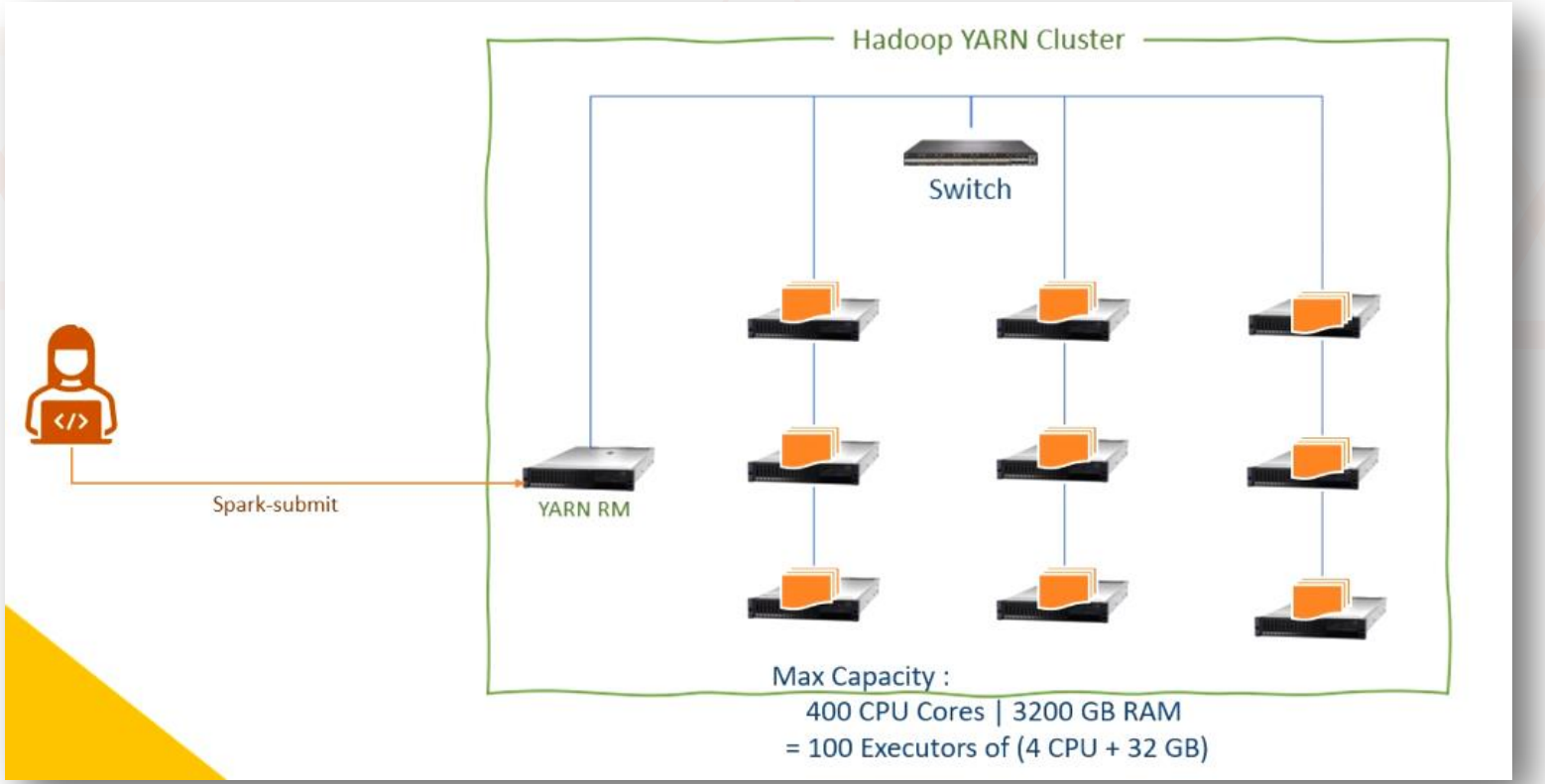


The second application will also sun on the same cluster. At any time, your cluster might be running more than one Spark application.
So in a typical case, a spark cluster is shared by multiple applications.

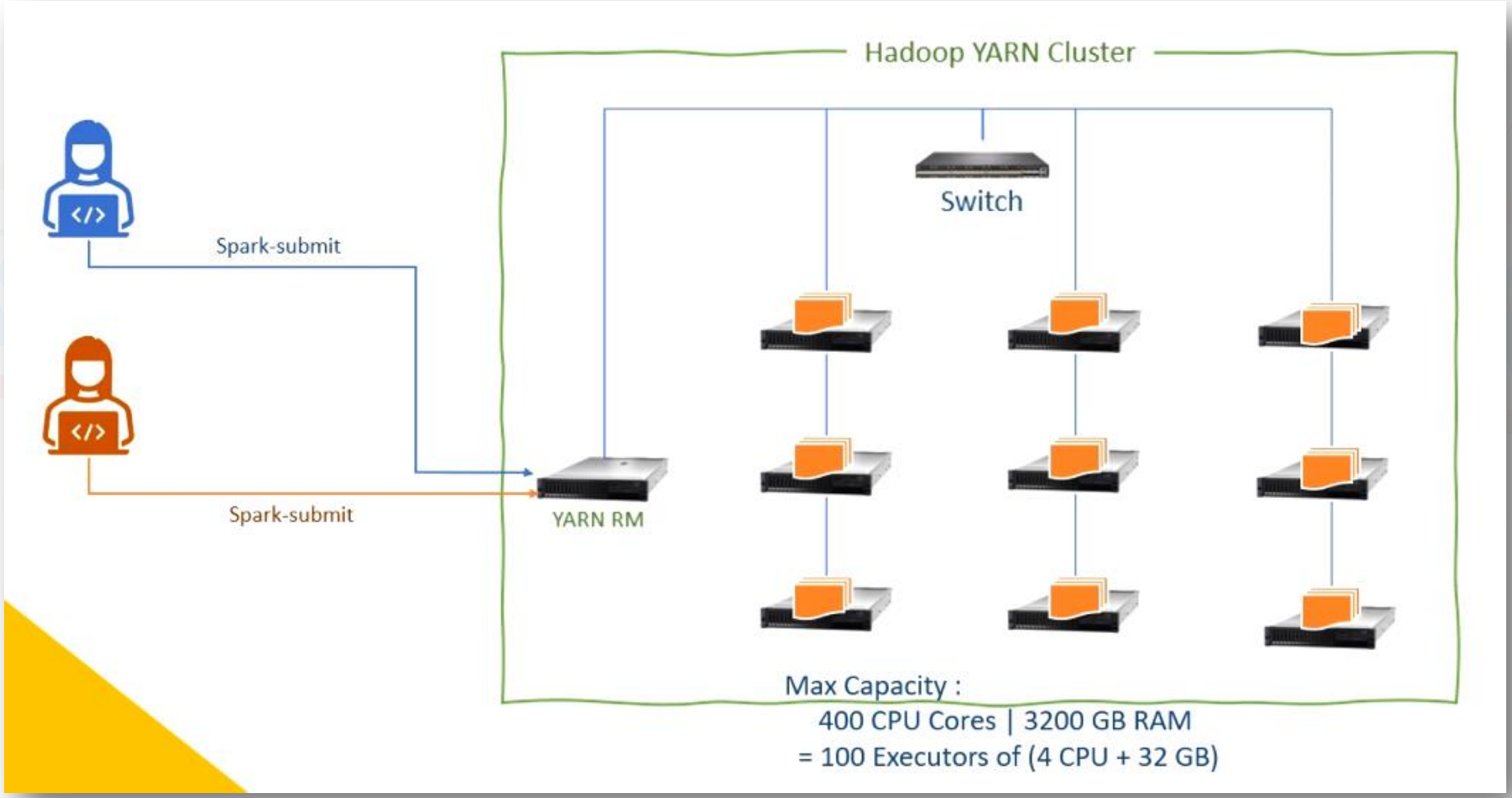


But the question is this: **How cluster-manager allocates resources to the Spark application, and when do they release those resources?**

Let's assume you have a cluster. And the maximum capacity of the cluster is to create 100 containers. So we cannot have more than 100 containers at a time. Now, you started one application, and the application demands 100 containers. The resource manager will allocate 100 containers to the first application.



What if we submit another application. The second application is small, and it needs only five containers. But we do not have any containers left. So the second application must wait for some resources to become free. That's not efficient. A small application has to wait for a large application to finish.



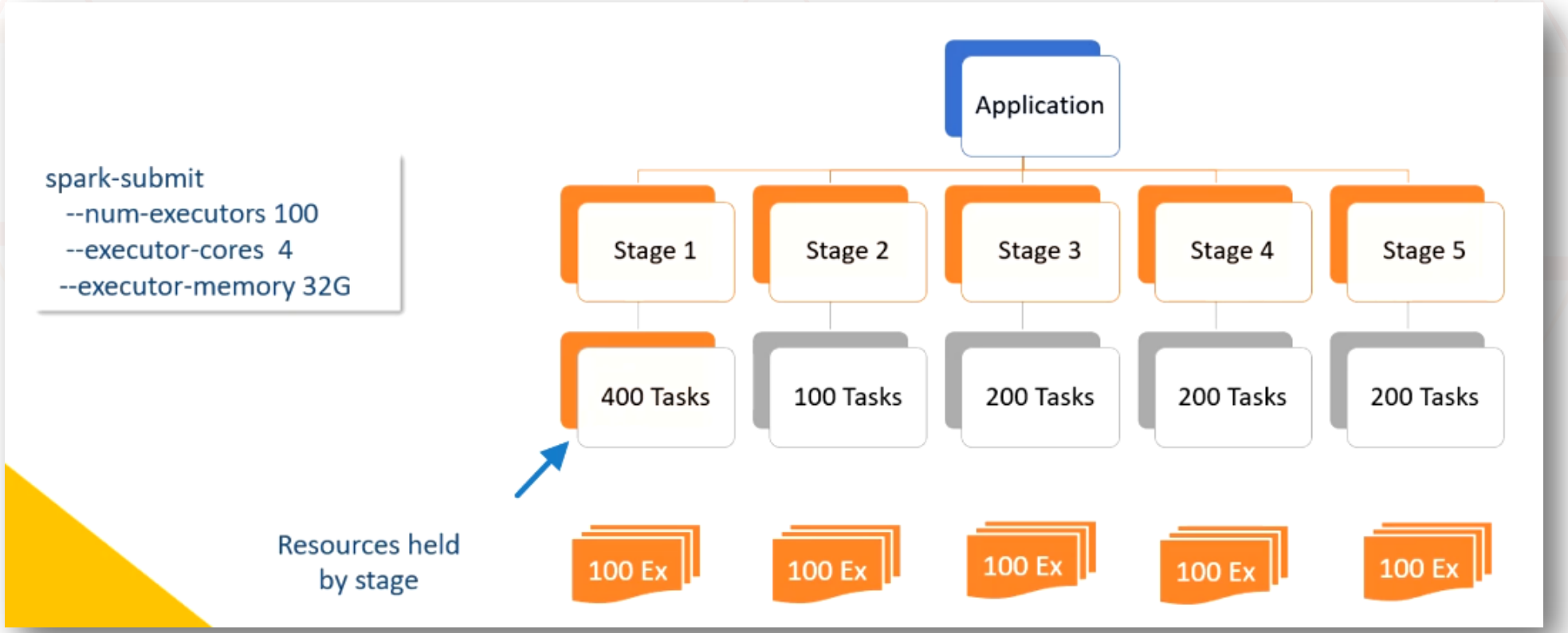
And this is an area where Spark offers you some customization so you can better manage your resources.
Spark offers you two resource allocation strategies:

1. Static Allocation
2. Dynamic Allocation

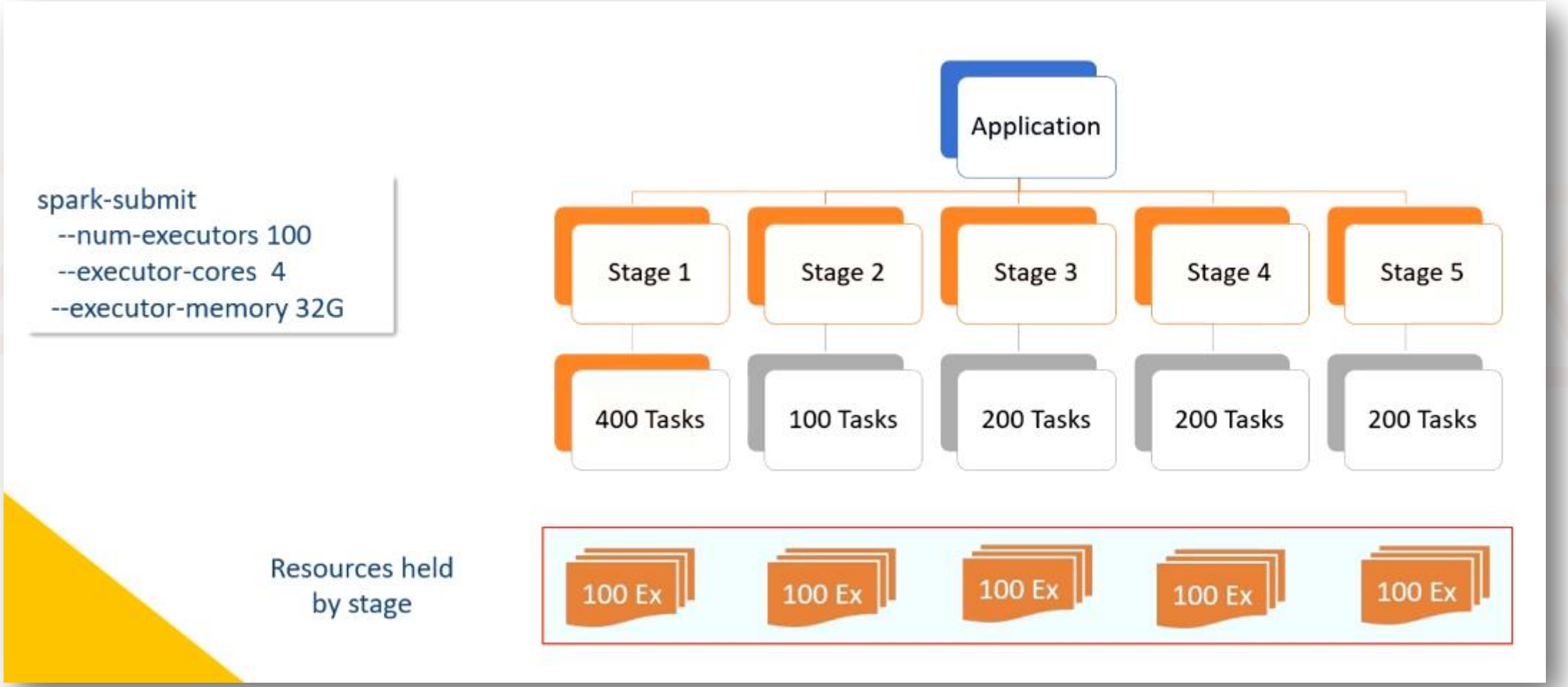
These two strategies are not for Cluster Resource managers.
These are for Spark. They do not have anything to do with the cluster resource manager.
These two approaches define how your application requests resources and how it releases the resources back to the cluster manager.

1. The static allocation is the default approach. In this approach, as soon as your driver starts, it will request executor containers from the cluster resource manager. The cluster manager gives as many resources as demanded by the driver. But now, your application will hold all those resources for the entire duration of running the application. Even if your application is not utilizing them, you will hold them with you. So the static allocation is a kind of first-come, first reserve approach. In this approach, the first application will demand resources and reserve them for the entire duration. These resources are released back to the cluster manager only when the application finishes its execution.

The static allocation approach has a problem of resource utilization. I mean, think about your application. You created an application that runs five stages. The first stage requires 400 parallel tasks, the second stage requires 100 parallel tasks, and all other stages require 200 parallel tasks. You demanded 100 executors with 4 CPU cores each. Why? Because you have a stage that needs to run 400 tasks. So you need 400 slots, and hence you demanded 100 executors, each with four slots. Your estimation is correct. You did the right thing.



But your application runs in five stages. Only one stage is making good use of 100 executors. The other four stages are not using all the executors. But you are holding all those executors till the end. That's a problem.

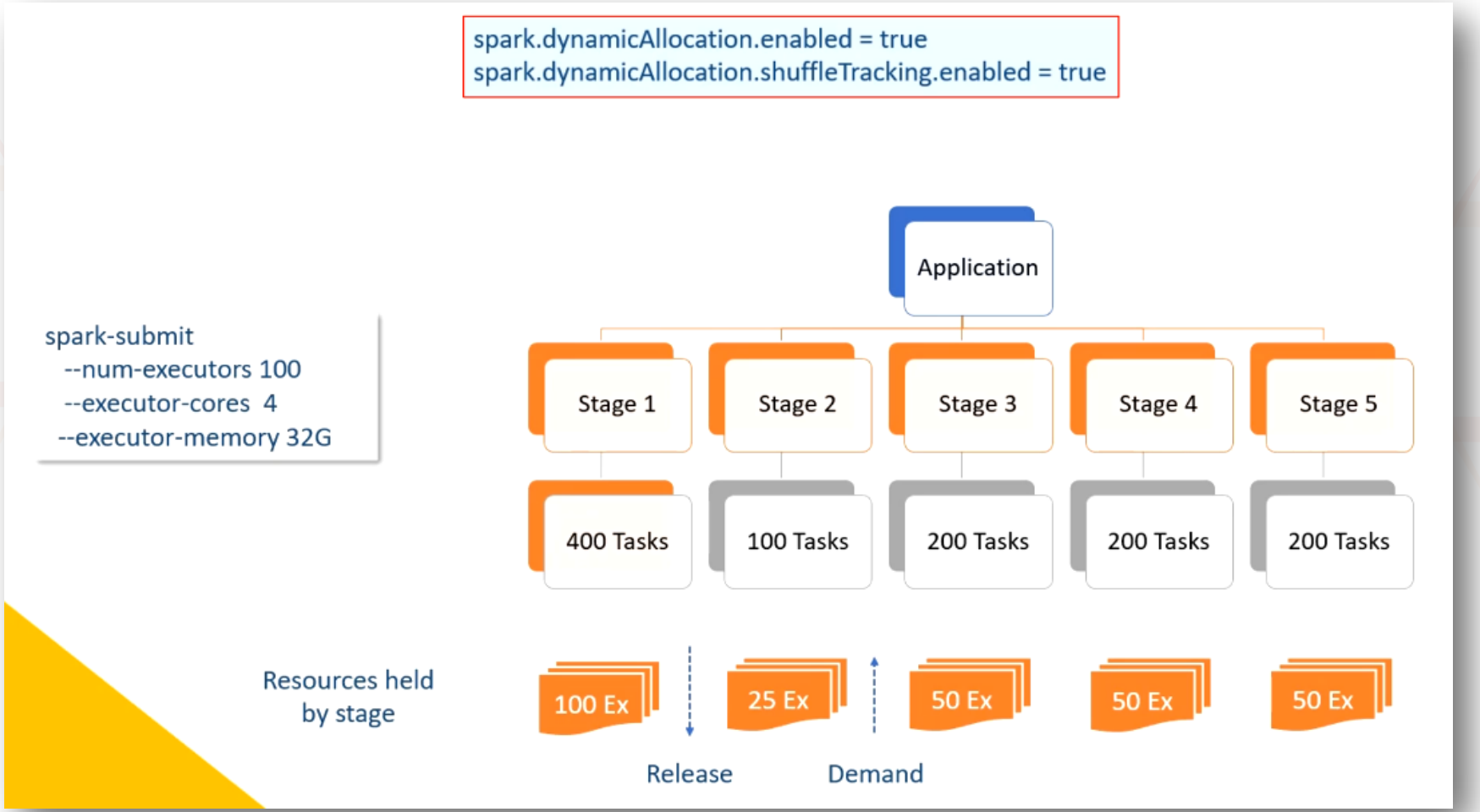


How do we solve this problem? That's where dynamic resource allocation comes into play.
Spark offers dynamic resource allocation, and you can enable it using the following configurations:

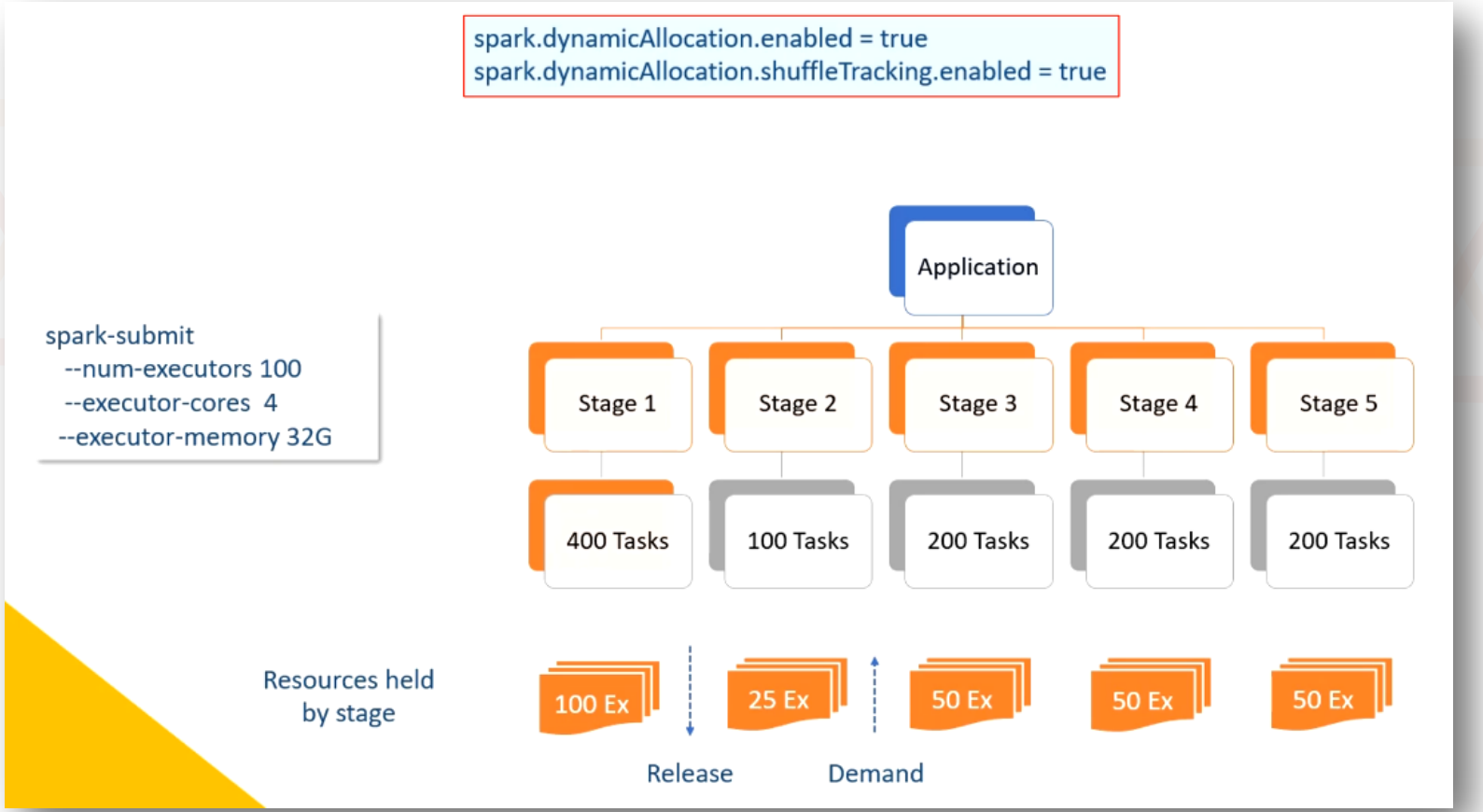
spark.dynamicAllocation.enabled
spark.dynamicAllocation.shuffleTracking.enabled

Dynamic allocation is disabled by default. But you can enable it to set these two configurations to true.
Once you enable dynamic allocation, your spark application will automatically release the executors if they are not using it. Similarly, they can again acquire executors when they are needed.
The resource manager has nothing to do here.
It is the Spark application that demands resources when needed and releases them if they are not required for the time being.
And your application starts doing it dynamically.

Now let me walk you through the same example.
So you need to run 400 parallel tasks in your first stage. So you will demand 100 executors.
The cluster manager will allocate your 100 executors, and you will start running your first stage.



Once you are done with your first stage, you need only 100 slots for the second stage. So you will release some executors back to the cluster manager. Why? Because you do not need them for the moment. You finished your second stage with fewer resources. But now you are into the third stage, and you need more slots. So your application will again demand more slots and take them from the cluster manager.



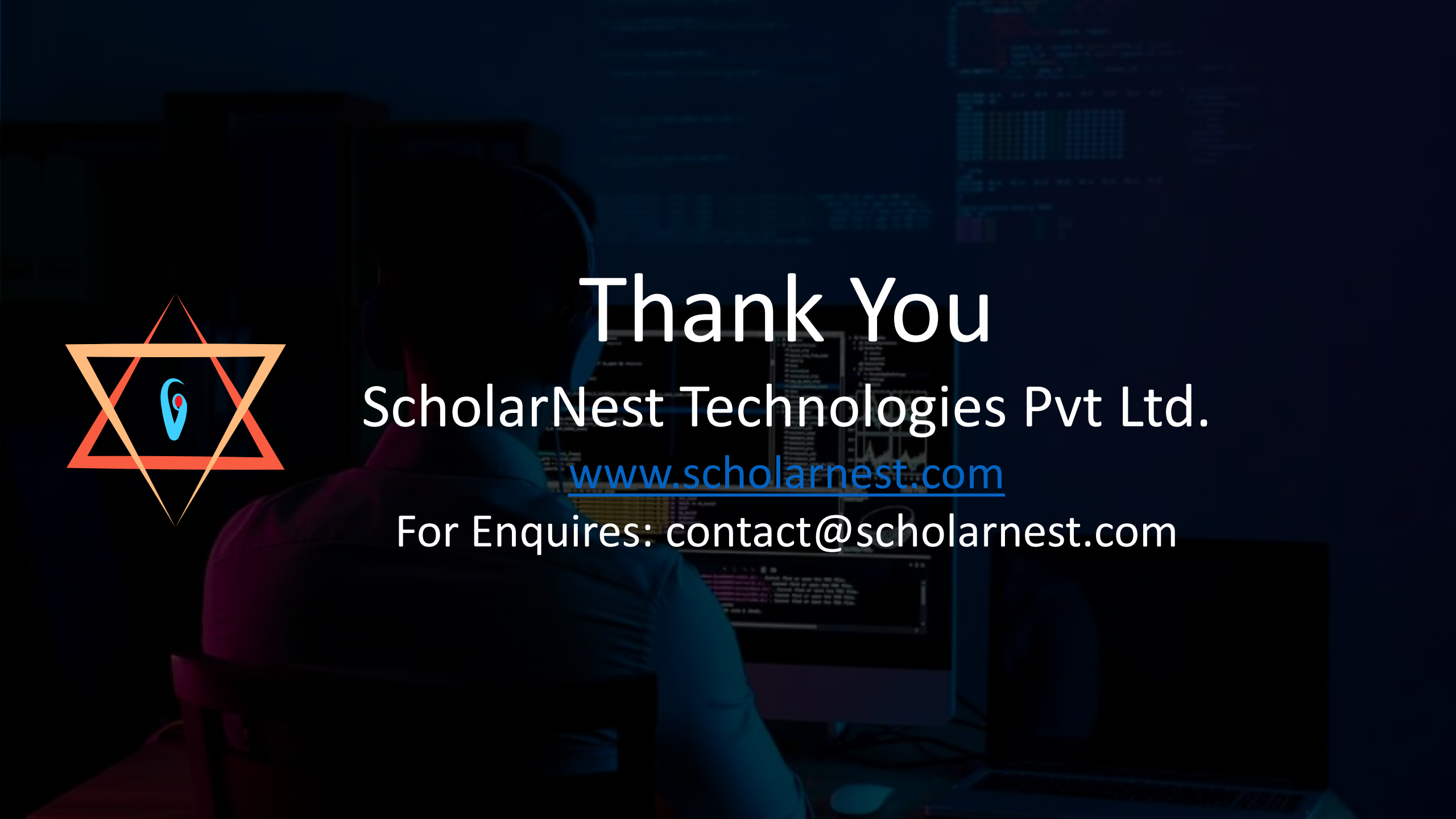
You have two more important configurations:

spark.dynamicAllocation.executorIdleTimeout = 60s
spark.dynamicAllocation.schedulerBacklogTimeout = 1s

The default value of idle timeout is 60 seconds.
So if an executor is idle and you do not have any task running on an executor for 60 seconds, your application will release the executor back to the cluster manager.
You can change this configuration, but 60 seconds is a reasonable value for a standard application.

The default value of this backlog timeout is 1 second.
So if you have some pending tasks for more than one second and do not have any free executor where you can allocate your pending task, your application will request more executors.
So the first configuration controls the release time for an idle executor.
And the second configuration controls the request time for more executors.

That's all for Spark Dynamic Resource allocation.
So we learned that the cluster manager schedules the Spark application driver, and resources for running the application are also provided by the resource manager.
Once your application driver starts, it can keep those resources or dynamically release and request them again.
If you have a shared cluster, you should enable dynamic allocation so your Spark applications can demand and release executors dynamically.



Thank You

ScholarNest Technologies Pvt Ltd.

www.scholarnest.com

For Enquires: contact@scholarnest.com

Spark Azure Databricks

Databricks Spark Certification and Beyond

Module:
Advanced Spark

Lecture:
Spark Schedulers

Spark Schedulers

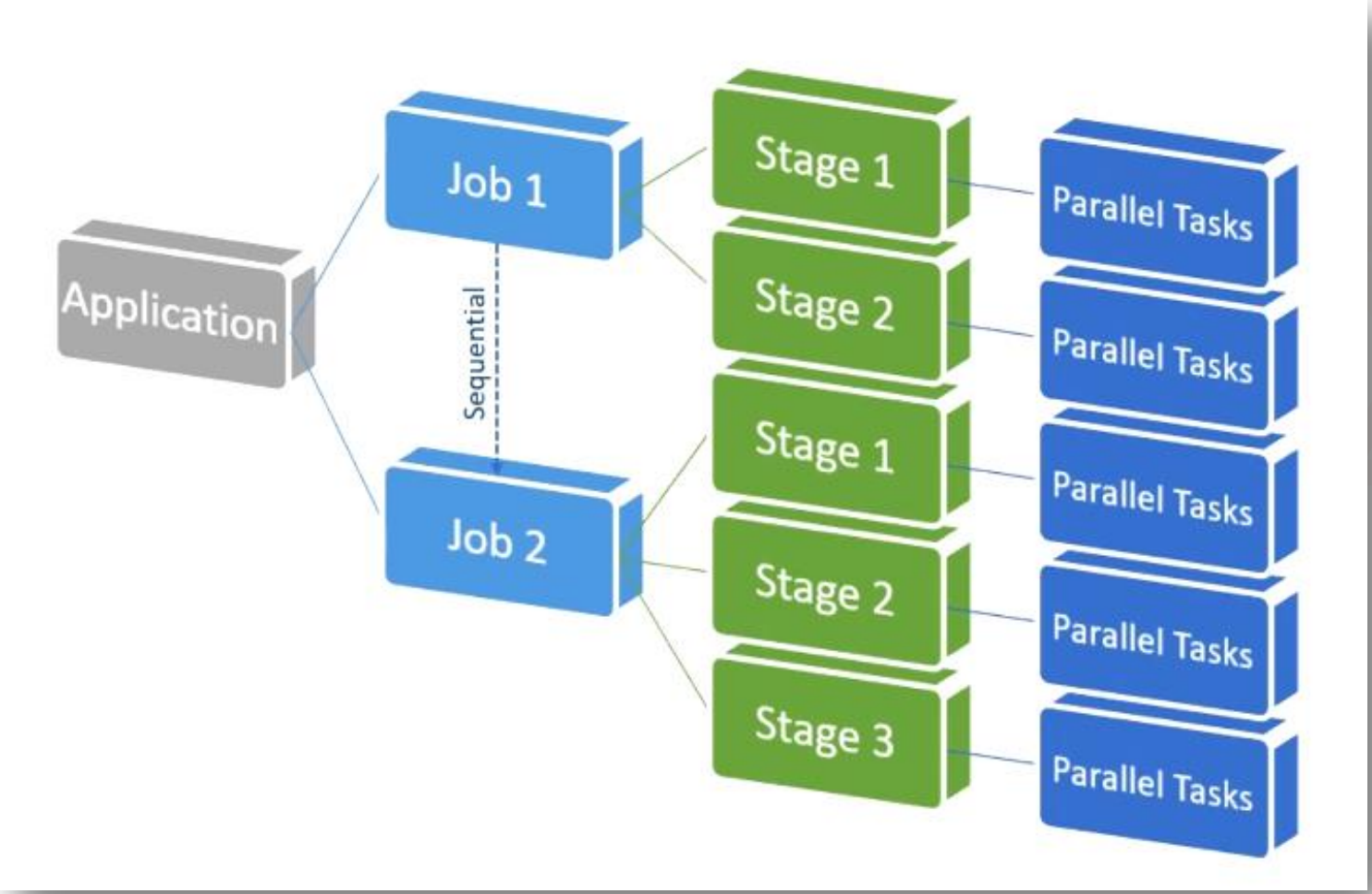
In the earlier chapter, I talked about the following types of Spark scheduling.

1. Scheduling Across Applications
2. Scheduling Within an Application

I already explained the first one.
In this video, I will talk about scheduling within a Spark application.

We already know that the Spark application runs multiple jobs.
Within a single Spark application, each of your actions triggers a spark job.

In a typical case, these spark jobs run sequentially. What does it mean?
I mean, the spark will start Job 1 and finish it. Then only Job 2 starts. That's what I mean by running Spark jobs in a sequence. However, you can also trigger them to run parallelly.



Here is an example code. What am I doing here? I am reading two data sets.
The first one is df1, and the second one is df2. Then I joined d1 with df2 and took the count.
(Reference: https://github.com/ScholarNest/PySpark-Examples/blob/main/fair-schedular/sequential-demo.py)

```
1   from pyspark.sql import SparkSession
2
3   if __name__ == "__main__":
4       spark = SparkSession \
5           .builder \
6           .appName("Demo") \
7           .master("local[3]") \
8           .config("spark.sql.autoBroadcastJoinThreshold", "50B") \
9           .getOrCreate()
10
11      df1 = spark.read.json("data/d1")
12      df2 = spark.read.json("data/d2")
13      print(df1.join(df2, "id", "inner").count())
14
15      df3 = spark.read.json("data/d3")
16      df4 = spark.read.json("data/d4")
17      print(df3.join(df4, "id", "inner").count())
18
```

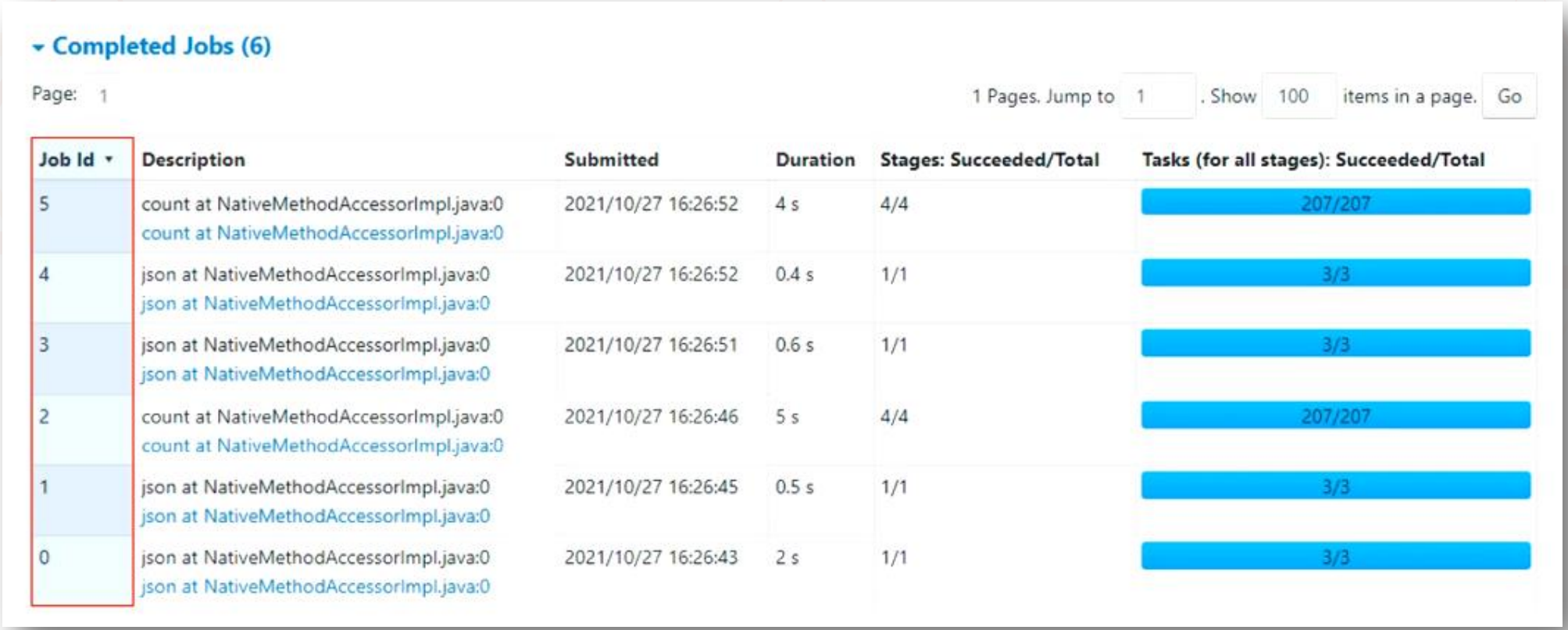

But I am also doing a similar thing with two more data sets. I am also reading Dataframe df3 and df4. I am again joining df3 with df4, taking the count, and printing it. I am doing two sets of independent operations. The first set of work will read df1, df2, join them together and trigger a count() action. Similarly, the second set of work will read df3, df4, join them together and trigger another count action.

```
1  from pyspark.sql import SparkSession
2
3  if __name__ == "__main__":
4      spark = SparkSession \
5          .builder \
6          .appName("Demo") \
7          .master("local[3]") \
8          .config("spark.sql.autoBroadcastJoinThreshold", "50B") \
9          .getOrCreate()
10
11     df1 = spark.read.json("data/d1")
12     df2 = spark.read.json("data/d2")
13     print(df1.join(df2, "id", "inner").count())
14
15     df3 = spark.read.json("data/d3")
16     df4 = spark.read.json("data/d4")
17     print(df3.join(df4, "id", "inner").count())
```


These two joins and counts are independent.
The first set of work has nothing to do with the second set of work.
They do not have any dependency on each other. But my code will run them in a sequence.

```
1   from pyspark.sql import SparkSession
2
3   if __name__ == "__main__":
4       spark = SparkSession \
5           .builder \
6           .appName("Demo") \
7           .master("local[3]") \
8           .config("spark.sql.autoBroadcastJoinThreshold", "50B") \
9           .getOrCreate()
10
11      df1 = spark.read.json("data/d1")
12      df2 = spark.read.json("data/d2")
13      print(df1.join(df2, "id", "inner").count())
14
15      df3 = spark.read.json("data/d3")
16      df4 = spark.read.json("data/d4")
17      print(df3.join(df4, "id", "inner").count())
```


I ran the previous code and here is the jobs section of Spark UI.
So we ran six jobs starting from job id zero to job id five. Now you look at the submitted timestamp. Job zero started at 16:26:43, and it took 2 seconds to complete. Job one started two seconds later when job zero finished. This guy started at 16:26:45, which is exactly two seconds of the run time of the job zero. If you look at the other jobs, they all started one after the other. So Spark will run your jobs in sequential order, one after the other.



Completed Jobs (6)

Page: 1 — 1 Pages. Jump to 1 . Show 100 items in a page. Go

Job Id	Description	Submitted	Duration	Stages: Succeeded/Total	Tasks (for all stages): Succeeded/Total
5	count at NativeMethodAccessorImpl.java:0 count at NativeMethodAccessorImpl.java:0	2021/10/27 16:26:52	4 s	4/4	207/207
4	json at NativeMethodAccessorImpl.java:0 json at NativeMethodAccessorImpl.java:0	2021/10/27 16:26:52	0.4 s	1/1	3/3
3	json at NativeMethodAccessorImpl.java:0 json at NativeMethodAccessorImpl.java:0	2021/10/27 16:26:51	0.6 s	1/1	3/3
2	count at NativeMethodAccessorImpl.java:0 count at NativeMethodAccessorImpl.java:0	2021/10/27 16:26:46	5 s	4/4	207/207
1	json at NativeMethodAccessorImpl.java:0 json at NativeMethodAccessorImpl.java:0	2021/10/27 16:26:45	0.5 s	1/1	3/3
0	json at NativeMethodAccessorImpl.java:0 json at NativeMethodAccessorImpl.java:0	2021/10/27 16:26:43	2 s	1/1	3/3

Now let's come back to our code. I have these two independent code blocks. I prefer to run these two blocks in parallel. Can we do it? Yes, we can.
All we have to do is create two parallel threads and trigger each of these blocks from two different threads. I have to change the code and implement multithreading in my application.

```
1   from pyspark.sql import SparkSession
2
3   if __name__ == "__main__":
4       spark = SparkSession \
5           .builder \
6           .appName("Demo") \
7           .master("local[3]") \
8           .config("spark.sql.autoBroadcastJoinThreshold", "50B") \
9           .getOrCreate()
10
11      df1 = spark.read.json("data/d1")
12      df2 = spark.read.json("data/d2")
13      print(df1.join(df2, "id", "inner").count())
14
15      df3 = spark.read.json("data/d3")
16      df4 = spark.read.json("data/d4")
17      print(df3.join(df4, "id", "inner").count())
18
```


Here is the new code. I created this do_job function. This function takes two input parameters. These parameters are nothing but the data file names. So I will read the first data file into df1, then the second data file into df2, join these two data frames and count it. That's what we were doing earlier in two code blocks. Now I have a common function for doing that.

```
1  from pyspark.sql import SparkSession
2  import threading
3
4
5  def do_job(f1, f2):
6      df1 = spark.read.json(f1)
7      df2 = spark.read.json(f2)
8      outputs.append(df1.join(df2, "id", "inner").count())
9
10
```

```
11 if __name__ == "__main__":
12     spark = SparkSession \
13         .builder \
14         .appName("Demo") \
15         .master("local[3]") \
16         .config("spark.sql.autoBroadcastJoinThreshold", "50B") \
17         .config("spark.scheduler.mode", "FAIR") \
18         .getOrCreate()
19
20     file_prefix = "data/d"
21     jobs = []
22     outputs = []
23
24     for i in range(0, 2):
25         file1 = file_prefix + str(i + 1)
26         file2 = file_prefix + str(i + 2)
27         thread = threading.Thread(target=do_job, args=(file1, file2))
28         jobs.append(thread)
29
30     for j in jobs:
31         j.start()
32
33     for j in jobs:
34         j.join()
35
36     print(outputs)
```


Then I come back to my main method and run this loop. The for loop runs twice.
Every iteration will create one thread. The target for the thread is the same do_job function.
But I am changing the file names for each iteration. So we are creating two threads here.
Finally, I will start both the threads and then wait for both the threads to complete.

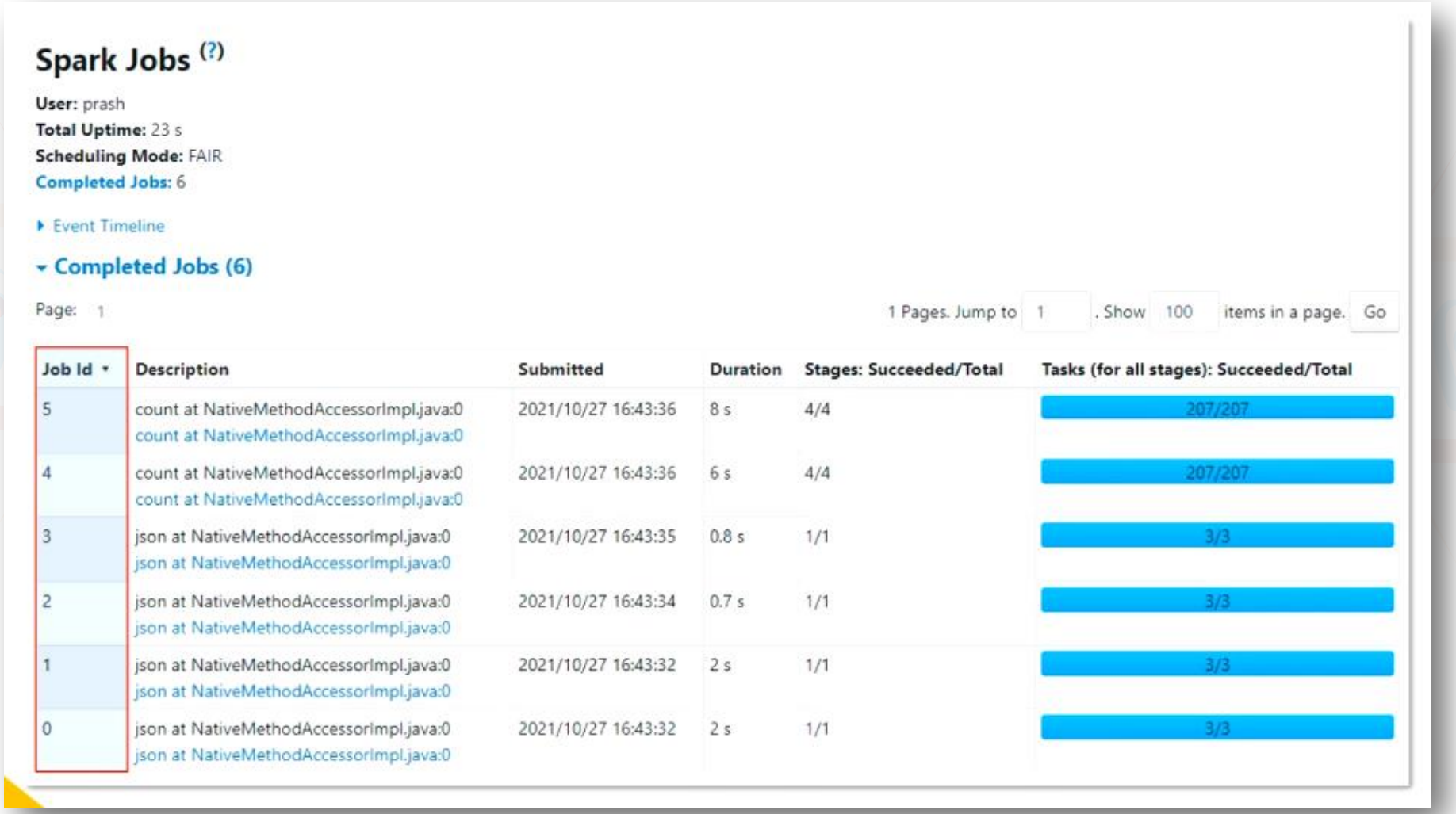
```python
from pyspark.sql import SparkSession
import threading


def do_job(f1, f2):
    df1 = spark.read.json(f1)
    df2 = spark.read.json(f2)
    outputs.append(df1.join(df2, "id", "inner").count())

```

```python
if __name__ == "__main__":
    spark = SparkSession \
        .builder \
        .appName("Demo") \
        .master("local[3]") \
        .config("spark.sql.autoBroadcastJoinThreshold", "50B") \
        .config("spark.scheduler.mode", "FAIR") \
        .getOrCreate()

    file_prefix = "data/d"
    jobs = []
    outputs = []

    for i in range(0, 2):
        file1 = file_prefix + str(i + 1)
        file2 = file_prefix + str(i + 2)
        thread = threading.Thread(target=do_job, args=(file1, file2))
        jobs.append(thread)

    for j in jobs:
        j.start()

    for j in jobs:
        j.join()

    print(outputs)
```


This code is designed to run the do_job function in two different threads, and both the threads will trigger a count() action.
Now let me run it and see what happens with my spark jobs.

```python
from pyspark.sql import SparkSession
import threading


def do_job(f1, f2):
    df1 = spark.read.json(f1)
    df2 = spark.read.json(f2)
    outputs.append(df1.join(df2, "id", "inner").count())


if __name__ == "__main__":
    spark = SparkSession \
        .builder \
        .appName("Demo") \
        .master("local[3]") \
        .config("spark.sql.autoBroadcastJoinThreshold", "50B") \
        .config("spark.scheduler.mode", "FAIR") \
        .getOrCreate()

    file_prefix = "data/d"
    jobs = []
    outputs = []

    for i in range(0, 2):
        file1 = file_prefix + str(i + 1)
        file2 = file_prefix + str(i + 2)
        thread = threading.Thread(target=do_job, args=(file1, file2))
        jobs.append(thread)

    for j in jobs:
        j.start()

    for j in jobs:
        j.join()

    print(outputs)
```


Here is your Spark UI. We still have six jobs starting from job id 0 to job id 5. But now, things are running in parallel. Job zero and job one started in parallel at the same time. Job four and job five also started in parallel at the same time.



Spark Jobs (?)

User: prash
Total Uptime: 23 s
Scheduling Mode: FAIR
Completed Jobs: 6

▸ Event Timeline

▾ **Completed Jobs (6)**

Page: 1 — 1 Pages. Jump to 1 . Show 100 items in a page. Go

Job Id	Description	Submitted	Duration	Stages: Succeeded/Total	Tasks (for all stages): Succeeded/Total
5	count at NativeMethodAccessorImpl.java:0 count at NativeMethodAccessorImpl.java:0	2021/10/27 16:43:36	8 s	4/4	207/207
4	count at NativeMethodAccessorImpl.java:0 count at NativeMethodAccessorImpl.java:0	2021/10/27 16:43:36	6 s	4/4	207/207
3	json at NativeMethodAccessorImpl.java:0 json at NativeMethodAccessorImpl.java:0	2021/10/27 16:43:35	0.8 s	1/1	3/3
2	json at NativeMethodAccessorImpl.java:0 json at NativeMethodAccessorImpl.java:0	2021/10/27 16:43:34	0.7 s	1/1	3/3
1	json at NativeMethodAccessorImpl.java:0 json at NativeMethodAccessorImpl.java:0	2021/10/27 16:43:32	2 s	1/1	3/3
0	json at NativeMethodAccessorImpl.java:0 json at NativeMethodAccessorImpl.java:0	2021/10/27 16:43:32	2 s	1/1	3/3

So, your Spark application runs as a set of spark jobs. Each Spark job represents an action.
You can submit these actions sequentially, or you can also submit these actions using parallel threads.
But when you run spark jobs in parallel, you have to think about resourcing.
I mean, How will spark allocate executor slots for these parallel jobs? Because each action or a job runs as a set of parallel tasks.
So each job requires some resources to run multiple tasks in parallel.
When you create a multithreaded spark application to submit two or more parallel jobs, they all need resources for their corresponding tasks.
And that causes competition between jobs to acquire resources.
How do we handle it?
This is what we mean by scheduling within an application.
If your application is a single-threaded spark application, you do not worry about the scheduling within the application.
But if you create a multithreaded Spark application and submit jobs from multiple parallel threads, job scheduling becomes an important part.

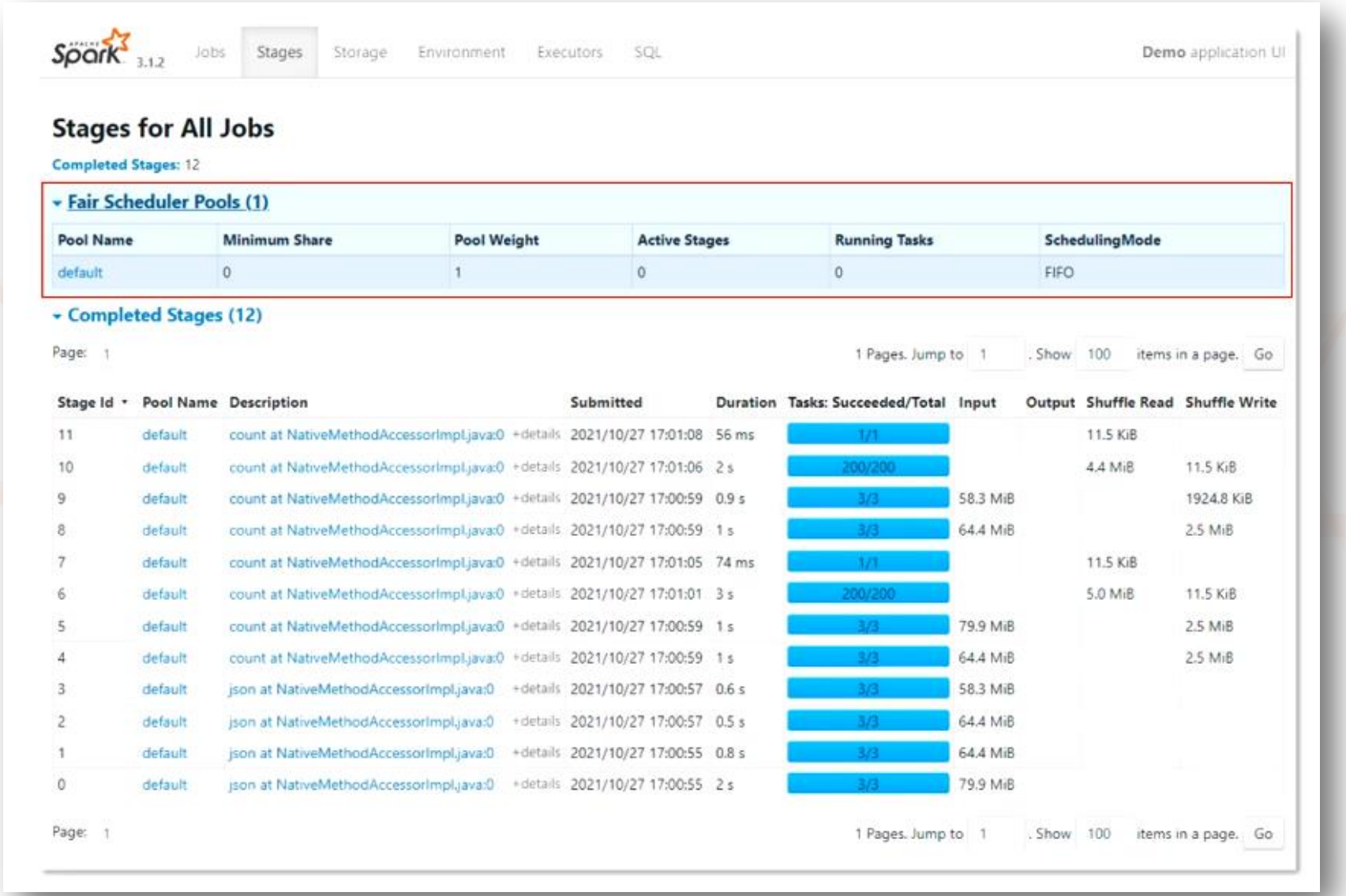
So how do we handle it?
By default, Spark's job scheduler runs jobs in a FIFO fashion.
Each job is divided into stages, and the first job gets priority on all available resources.
So if the first job has some stages and tasks, it will consume as many resources as needed.
Then the second job gets priority.
So the point is straight.
If the jobs at the queue head don't need to use some resources, the next job can start to run.
But if the jobs at the head of the queue are large, then later jobs may be delayed significantly.

However, you can change the FIFO scheduler and configure spark to use the FAIR schedular.
Here is the configuration:

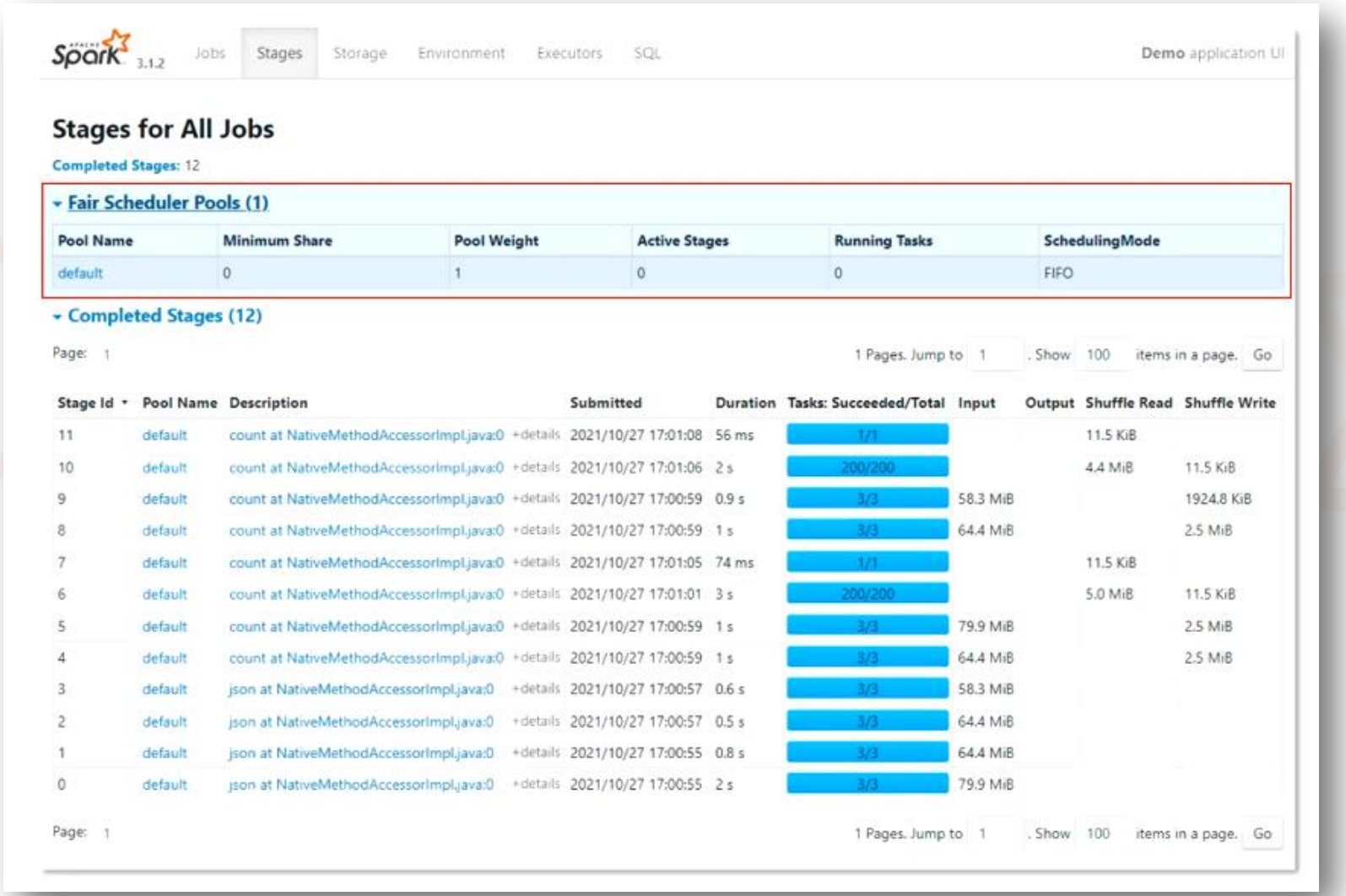
spark.scheduler.mode = FAIR

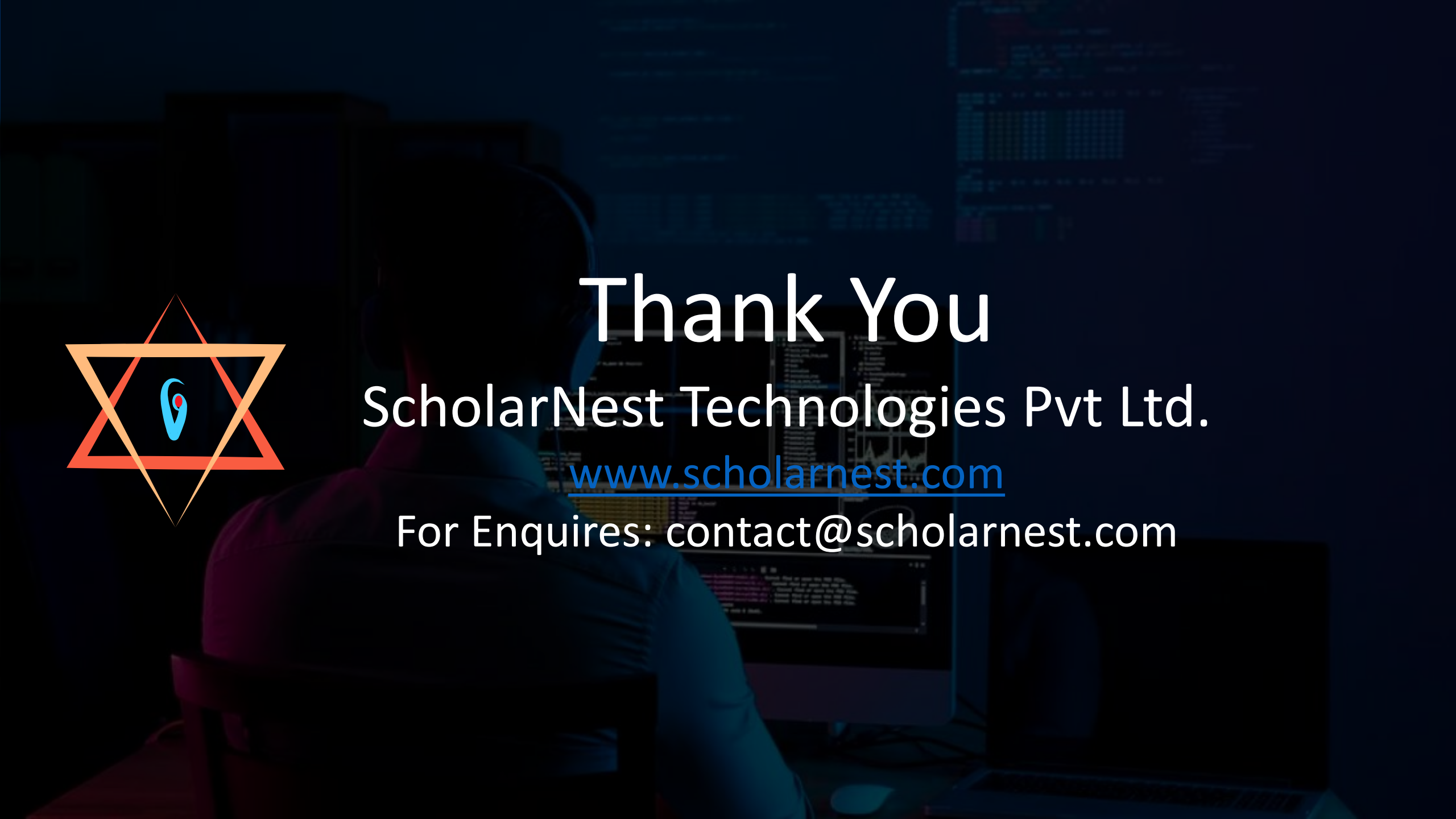
Under fair sharing, Spark assigns tasks between jobs in a round-robin fashion.
What does it mean? Simple! Assign one task in a slot from the first job, then assign one task in a slot from the second job, and so on.
In this approach, no parallel jobs will be waiting for resources, and all get a roughly equal share of cluster resources.

Once you enable FAIR schedular, you can see the list of fair scheduler pools in your spark UI. Here is an example screenshot. You can see it here. We are using the FAIR schedular default pool.



And within the pool, we have a FIFO schedular. You can go further down to create and configure multiple pools within the FAIR schedular, but that may not be necessary.
Fair schedular with default pool works well enough.





Thank You

ScholarNest Technologies Pvt Ltd.

www.scholarnest.com

For Enquires: contact@scholarnest.com