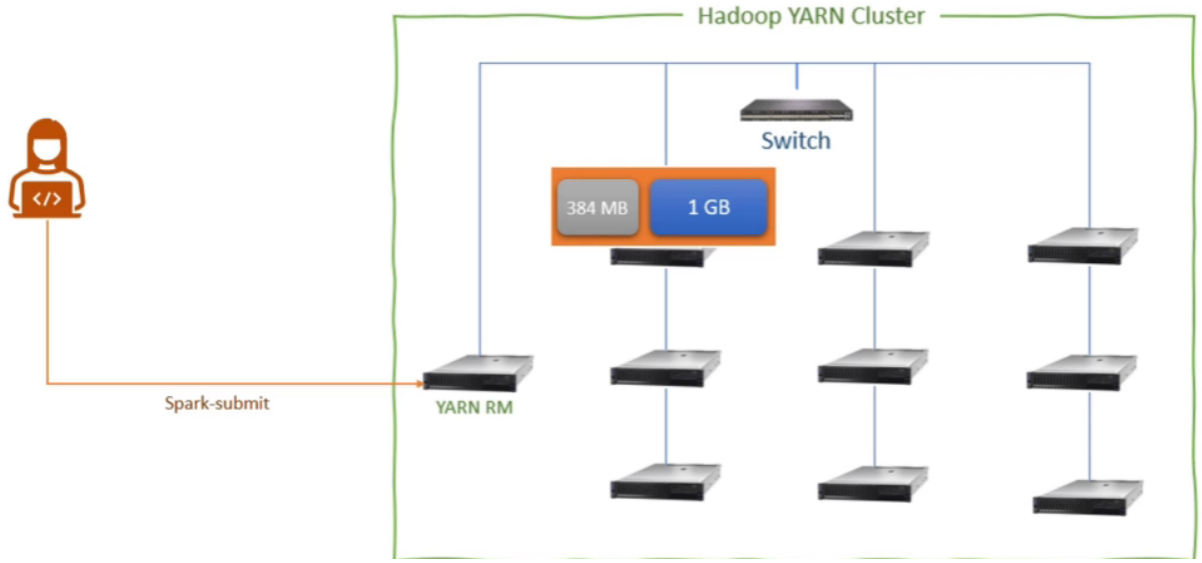
SPARK MEMORY ALLOCATION:

Assume we submitted spark application in a YARN cluster. YARN RM will allocate an application master(AM) container and start the driver JVM in the container. Driver will start with some memory allocation which we requested. We can ask for driver memory using 2 configurations.

1) spark.driver.memory 2) spark.driver.memoryOverhead

Let's assume we asked for spark.driver.memory = 1GB and default value for spark.driver.memoryOverhead is 10% =0.10 . Yarn RM will allocate 1GB of memory for driver JVM. And it will also allocate 10% of requested memory (384 MB) . we asked for spark driver memory as 1GB. 10% of spark driver memory is 100MB. 100MB is less than 384MB. So, YARN RM will allocate 384MB for overhead. Now total memory of container is 384MB + 1GB . What is the purpose of 384MB overhead memory ? overhead memory is used by the container process or any other non JVM process within the container. Our spark driver uses all JVM heap but nothing from the overhead.

1. spark.driver.memory = 1GB -> JVM Memory
2. spark.driver.memoryOverhead = 0.01 -> max(10% or 384 MB)



That's all about driver memory allocation.

Now driver is started with 1GB of JVM heap. So, driver will again request for executor containers from YARN. YARN RM will allocate bunch of executor containers. But how much memory will we get for each executor container? The total memory allocated to executor container is sum of the overhead memory and Heap Memory and off heap memory and pyspark memory.

So, spark driver will ask for executor container memory using 4 configurations as shown below.

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, driver will look at all these configurations to calculate our memory requirement and summed up. Let's say we asked spark executor memory as 8GB. Default value of spark executor memory overhead is 10% .

Let's say other 2 configurations are not set, and default value is 0. So how much memory do we get for our executor container? 8800 MB.(8GB +800MB)

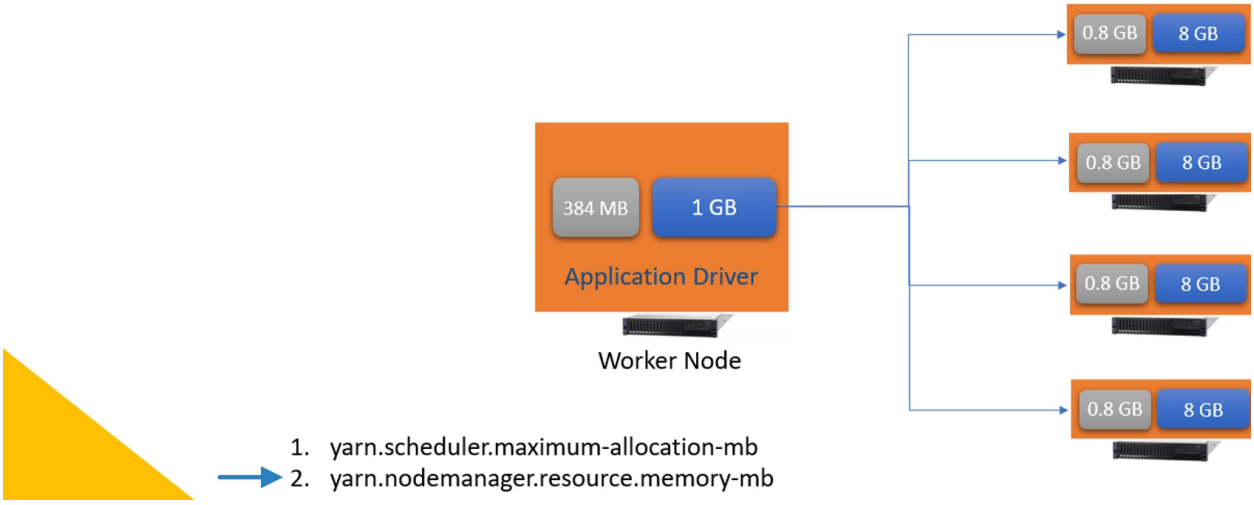
1. Overhead Memory -> spark.executor.memoryOverhead = 0.1
2. Heap Memory -> spark.executor.memory = 8GB
3. Off Heap Memory -> spark.memory.offHeap.size = 0
4. PySpark Memory -> spark.executor.pyspark.memory = 0

Total container memory now comes to 8800MB. So, driver will ask for 8.8GB containers to the YARN RM. But do we get an 8.8GB container? That depends on our cluster configuration. The container should run on a worker node in the YARN cluster. But if worker node is 6GB machine, YARN cannot allocate an 8GB container on a 6GB machine. Because there is not enough physical memory. So, before we ask for driver/executor memory, we should check with our cluster admin for the maximum allowed value.

If we are using YARN RM, we need to look for below configurations.

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

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



For example, if we asked for 4GB spark driver memory means we will get 4GB JVM heap and 400MB off JVM overhead memory. Now we have 3 limits. Our spark driver JVM cannot use more than 4GB. Our non JVM workload in the container cannot use more than 400MB and our container cannot use more than 4.4GB of memory in total. If any of these limits are violated , we will see an Out of Memory(OOM) exception.

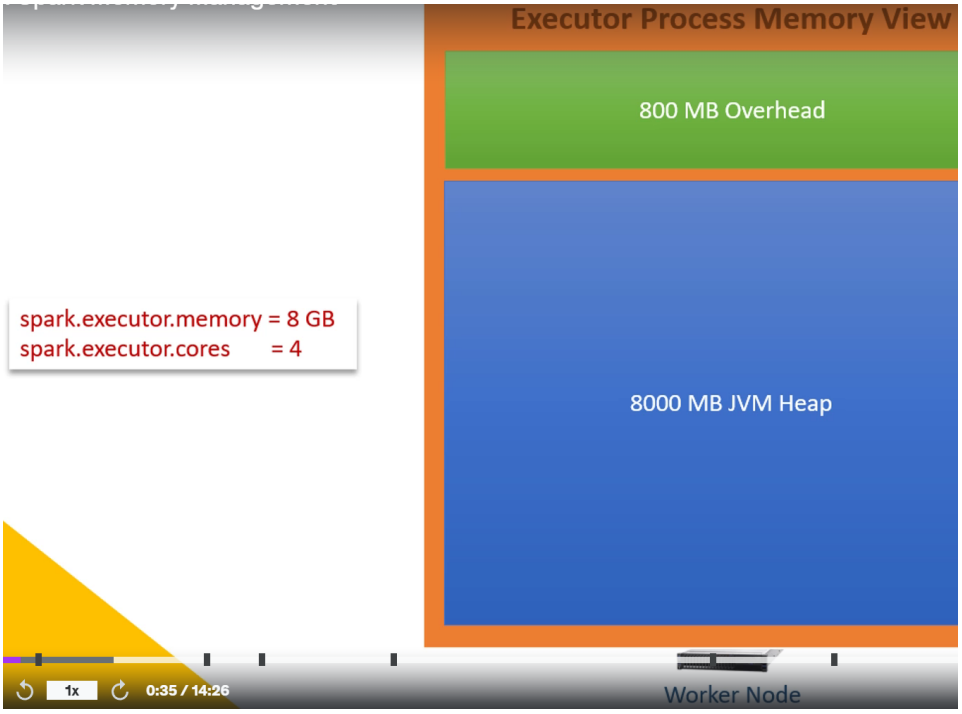
1. What is the Physical memory limit at the worker node?
   yarn.scheduler.maximum-allocation-mb
2. What is the PySpark executor memory?

How much memory do we get for pyspark ? pyspark is not a JVM process. So, we will not get anything from spark executor JVM memory and spark executor non JVM memory overhead. Some 300-400MB of this is constantly consumed by the container process and other internal process. So pyspark will get approximately 400MB. So we have 1 more limit. If our pyspark consumes more than what can be accommodated in the overhead, we can see an OOM error.

Spark container memory allocation looks like below.

We have a container and container has got some memory. This total memory is broken to 2 parts. Heap(driver/executor memory) and overhead memory(OS memory). Heap memory goes to our JVM. We call it driver memory when we are running a driver in this container. lly we call it executor memory when the container is running an executor. Overhead memory is used for network buffers. We will be using overhead memory for our shuffle exchange or reading partition data from remote storage etc.

SPARK MEMORY MANAGEMENT

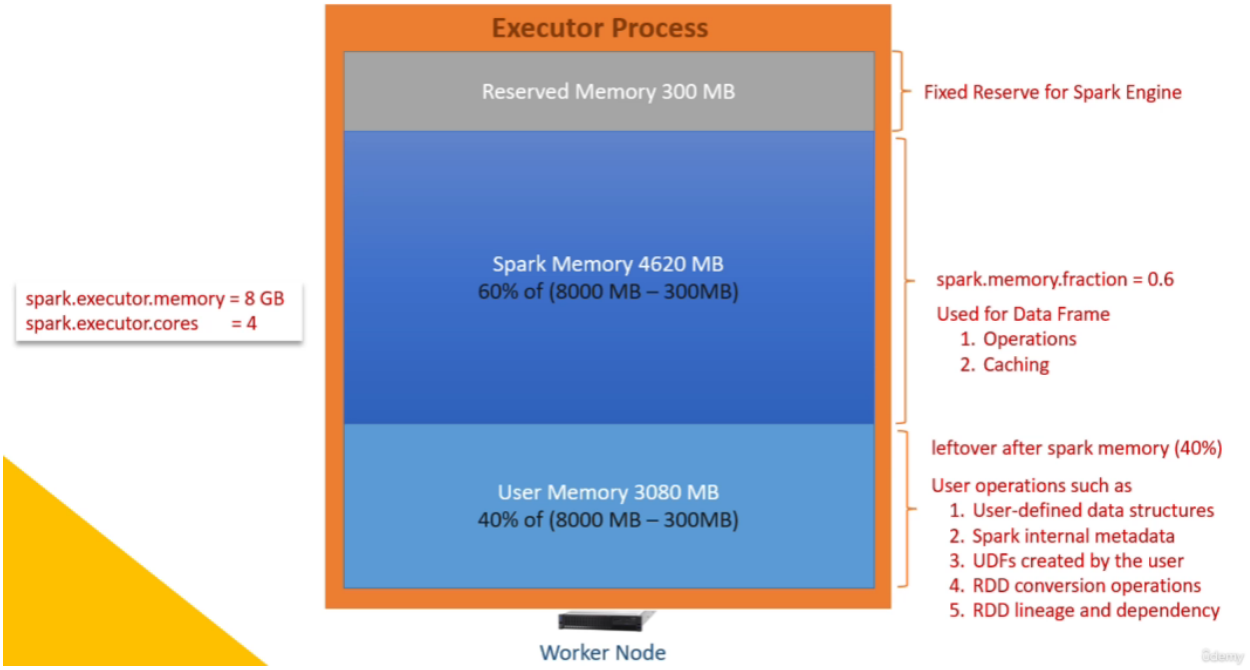


Container memory in worker node is splitted as heap memory and overhead memory.

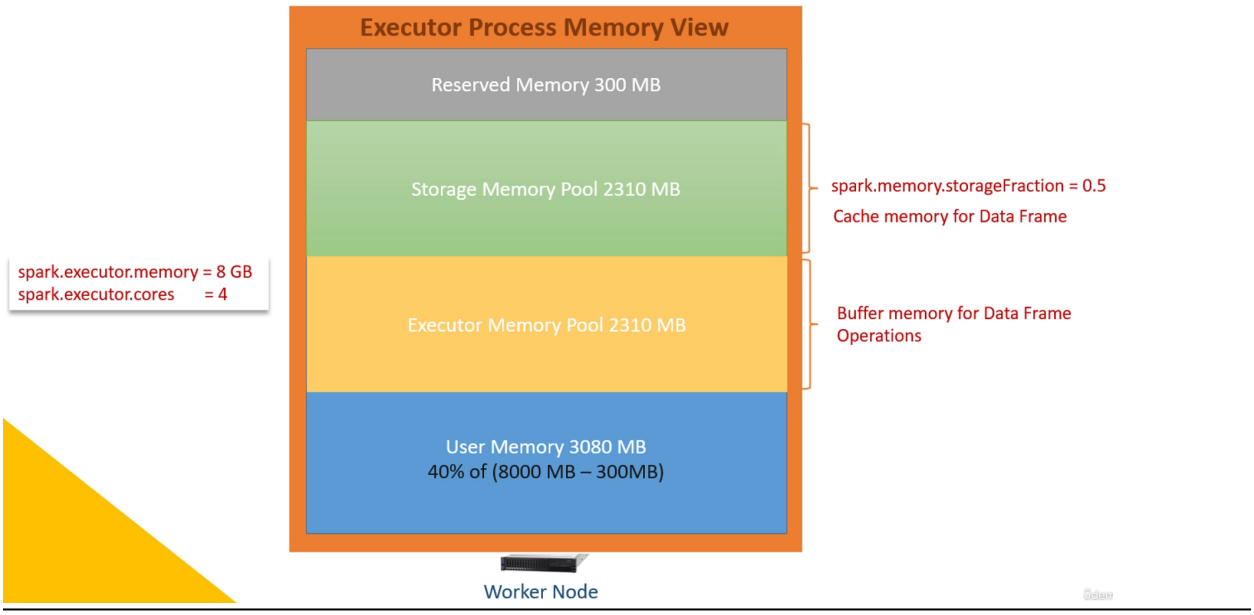
Now **heap memory is further broken down to 3 parts. Reserved Memory, Spark Memory, and User Memory.** Let's assume I got 8GB memory for JVM heap. Now this will divide in 3 parts. Spark will reserve 300MB for itself and it is fixed. Spark engine will itself use this 300 MB.

We can ask for executor memory using spark.executor.memory configuration. We asked for 8GB and got the same for JVM heap. Spark will reserve 300MB and left with 7700MB. This remaining memory is again divided into 60-40 ratio. 60% goes to spark memory pool and remaining goes to user memory pool. If we want to change this 60-40 ratio, we can change using spark.memory.fraction configuration. However, default value is 60%. Spark memory pool is our main executor memory pool used for data frame operations and caching. User memory pool used for non-data frame operations. Here are some examples. If we create user defined data structures such as hash maps , spark would use user memory pool. Spark internal metadata and user defined functions are stored in user memory. All RDD operations and its information are performed in user memory. But if we are using data frame operations, they do not use user memory even if dataframe is internally translated and compiled into RDD. We will be using user memory only if you apply RDD operations directly in our

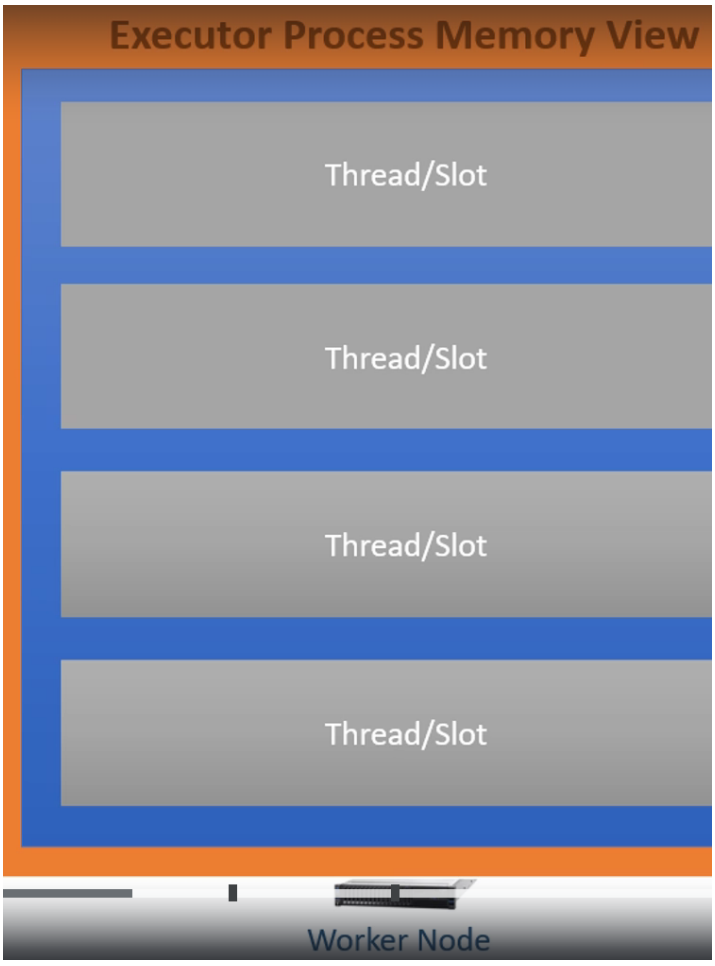
code. Spark memory pool is where our data frames and data frame operations live. We can increase from 60-70% or even more if we are not using UDF's , custom data structures and RDD operations. But we cannot make user memory to 0 or reduce it too much because we need it for metadata and other internal things.



Spark memory pool further broken down to 2 sub pools. Storage Memory and Executor memory. Default breakup is 50% each. But we can change it using spark.memory.storageFraction configuration. Default value of spark.memory.storageFraction configuration is 50%. So, spark will reserve 50% of memory pool for storage memory. Remaining 50% comes to executor memory. We use storage memory for caching data frames. Executor memory is for performing data frame computations. If we are joining 2 data frames, spark will need to buffer some data for performing joins. That buffer will be created in executor pool. If we are aggregating/performing some calculation, the required buffer memory comes from executor pool. Executor pool is short lived and we will use it for execution, and free it immediately as soon as our execution is complete. Storage pool is used to cache data frames. If we are using data frame cache operation,we will be caching data in the storage pool. So, storage pool is long term,we will cache the dataframe and keep it there as long as executor is running or you want to uncache it.

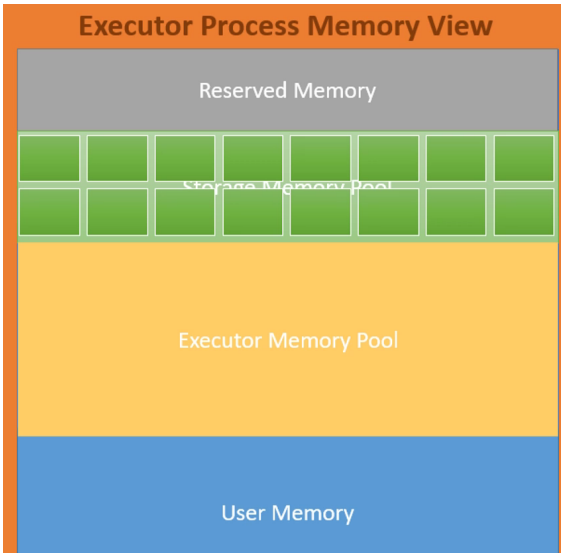


Our executor will have 4 slots, and we can run 4 parallel tasks in these slots, Slots are threads with in the same JVM. We have single JVM and all these slots are simply threads in the same JVM.

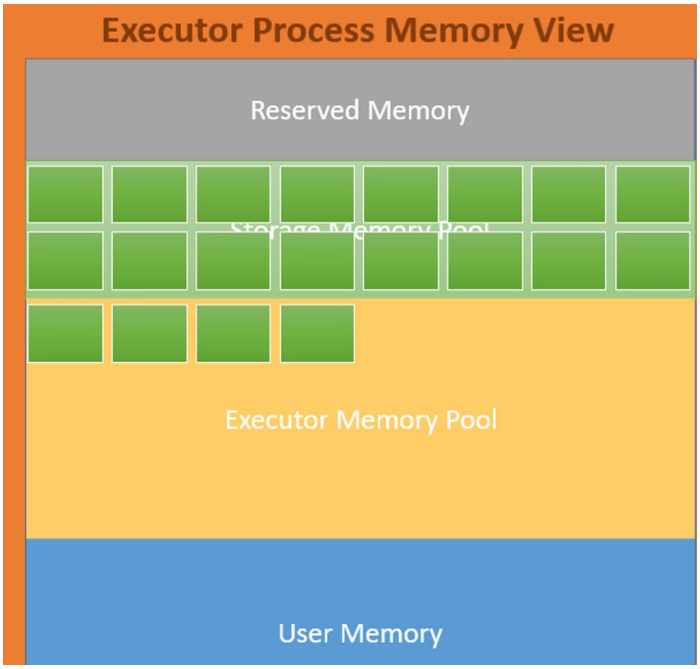


Let's say we have 1 executor JVM,2310 MB storage pool, another 2310MB executor pool and 4 threads to share these 2 memory pools. How much executor memory will each task get ? 2310/4 .

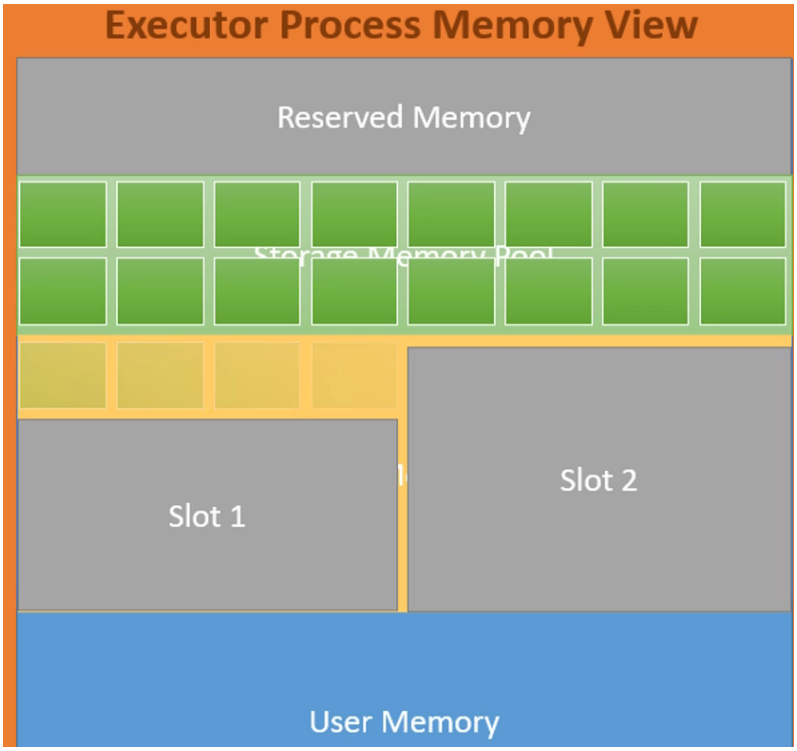
Let's say I have 4 slots but I have only 2 active slots available, so unified memory manager can allocate all the available execution memory amongst 2 active tasks. There is nothing reserved for any task. Task will demand execution memory and unified memory manager will allocate it from the pool. If executor memory pool is full, memory manager can also allocate executor memory from the storage memory pool as long as we have some free space.

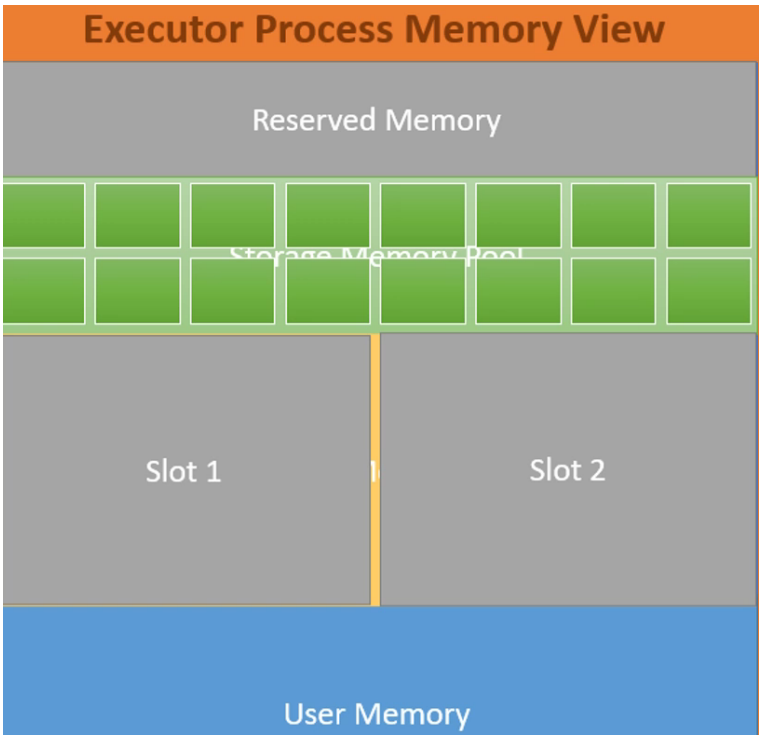


Let's assume we cached some data frames and entirely consumed storage memory pool. But want to cache some more data . So, memory manager will give us some more memory from executor memory pool. Executor memory was free so we consumed it.



Now executor is performing some join operation and it needs memory, so executor will start consuming memory as longas there is free space. But in the end memory manager will evict the cached data frames as shown in image 1 and spill them to disk to make more space for the executor as shown in 2nd image. We borrowed it from executor pool and now executor needs it. So, memory manager will evict it.





However, we reached boundary and executor needs more memory. This boundary is rigid and memory manager cannot evict any cached data blocks(from storage memory as shown above) beyond this boundary. If storage memory was free , memory manager should have given it to executor. But it is not free we already occupied . However, executor needs more memory. How to manage it? Executor will try to spill few things to the disk and make some more room for itself. If we cannot spill to disk, we will get OOM exception.

We have following configurations to control spark management.

Spark Memory Configurations

1. spark.executor.memoryOverhead
2. spark.executor.memory
3. spark.memory.fraction
4. spark.memory.storageFraction
5. spark.executor.cores

spark.executor.memoryOverhead will use us 10% extra of whatever we are asking for JVM and it is outside JVM and reserved for overhead.

spark.executor.memory → Our JVM Memory comes from spark.executor.memory. This configuration allow us to ask for executor JVM Memory. Whatever we get, spark engine will reserve 300MB from our allocation.

spark.memory.fraction → Now we can configure spark.memory.fraction to tell how much we want to use for data frame operations. Default value is 60% . we can also increase it. Left over is kept aside for non-dataframe operations and we call it as user memory.

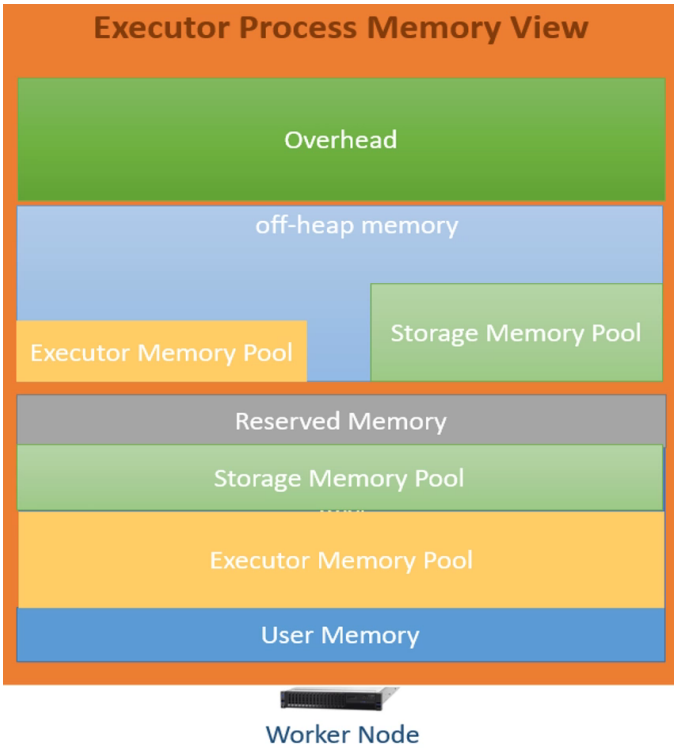
spark.memory.storageFraction → allows us to set hard boundary that memory manager cannot evict. If we are not dependent on cache data, we can reduce this hard limit.

spark.executor.cores → This configuration defines maximum number of concurrent threads. If you have too many threads, you will have too many tasks competing for memory. If you have single thread, you might not be able to use the memory efficiently. In general spark recommends two or more cores per executor but should not go beyond 5 cores. More than 5 cores cause excessive memory management overhead . so, they recommend starting at 5 cores.

These below 3 configurations will give us off-heap memory outside our JVM.

6. spark.memory.offHeap.enabled
7. spark.memory.offHeap.size
8. spark.executor.pyspark.memory

Most of the spark operations and data caching are performed in JVM heap and they perform best when using on-heap memory. However, JVM heap is subject to garbage collection.so if you are allocating huge amount of heap memory to our executor you might see excessive garbage collection delays. However, spark 3.X was optimized to perform some operations in off-heap memory. Using off -heap memory gives us flexibility of managing memory by ourself and avoiding Garbage Collection delays. Spark can take advantage of this idea and can use off-heap memory. So, if we need an excessive amount of memory for our spark application, it might help to take some off-heap memory. For large memory requirements, mixing some on-heap and off-heap might help to reduce GC delays. By default , off heap memory feature is disabled. You can enable it by setting **spark.memory.offHeap.enabled =true** . and we can set our off-heap memory requirement using spark.memory.offHeap.size . Spark will use off-heap memory to extend the size of spark executor memory and storage memory. Off heap memory means 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 executor and storage memory pools.



Scala is a JVM language and hence spark is also JVM application. But if we are using pyspark, your application may need python worker. These python workers cannot use JVM heap memory. So, they use off heap overhead memory. If we need more off-heap overhead memory for our python workers, we can set that extra memory requirement for our python workers using spark.executor.pyspark.memory. we do not have 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 doesn't set a default value for this configuration parameter. But if you need some extra memory for our python workers, we can set requirement using spark.executor.pyspark.memory

SPARK ADAPTIVE QUERY EXECUTION- DYNAMICALLY COALESCING SHUFFLE PARTITIONS

Spark Adaptive Query Execution offers following features

1. Dynamically coalescing shuffle partitions
2. Dynamically switching join strategies
3. Dynamically optimizing skew joins

Let's say we are running simple groupby query in our spark code.

call_id	call_duration	tower_location
10001	6	bangalore_tower_a
10002	12	bangalore_tower_b
10003	15	bangalore_tower_b
10004	28	bangalore_tower_a

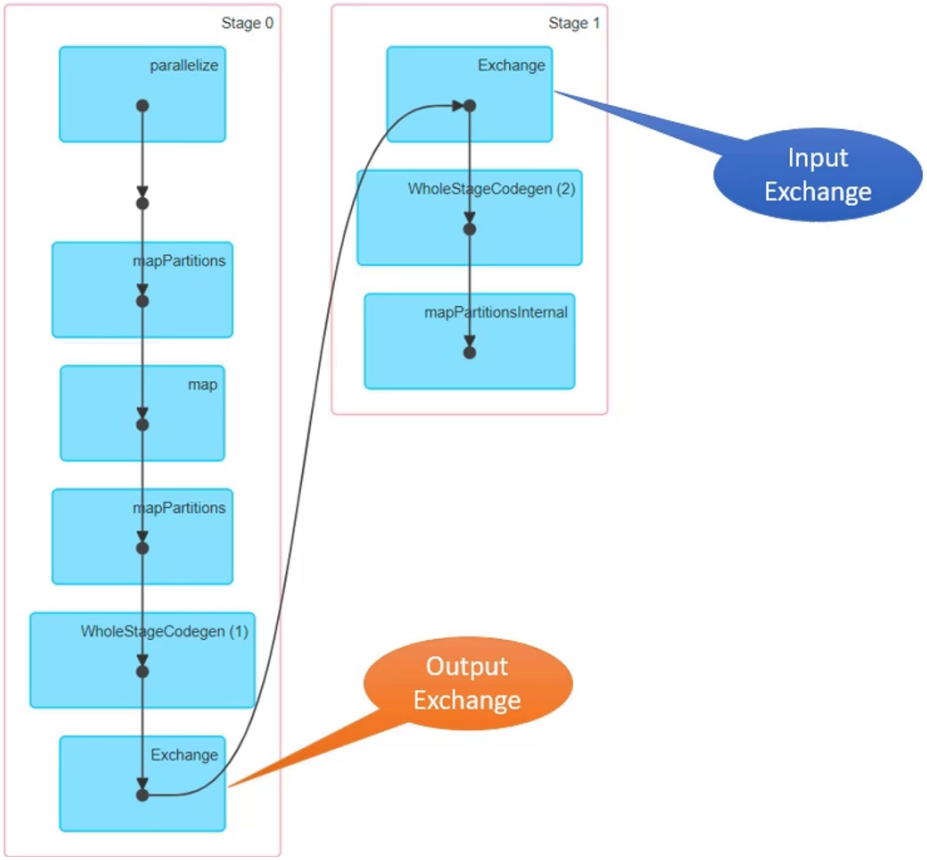
```
SELECT tower_location,
       sum(call_duration) as duration_served
FROM  call_records
GROUP BY  tower_location;
```

```
df.groupBy("tower_location")
  .agg(sum("call_duration").alias("duration_served"))
```

In calls_records table, this table stores information about cell phone calls made by different users.

How will spark execute this query? Spark will take the code, create an execution plan for the query and execute it on cluster. This spark job will have a 2-stage plan. Execution plan for the query shown below, This below actual plan for the large volume of data might not look same. Stage 0 reads data from the source table and fills it to output exchange. The second stage will read data from output exchange and brings it to the input exchange. This process is called shuffle-sort. Why do we have

shuffle/sort in our execution plan? Because we are doing groupBy operation and groupBy is a wide dependency transformation. So, we are likely to have a shuffle/sort in our execution plan.Stage 1 is dependent on stage 0 . So, stage 1 cannot start unless stage 0 is complete.



Output for query:

```
tower_location          duration_served
--------------------------------------
bangalore_tower_a                34
bangalore_tower_b                27
--------------------------------------
```

Input exchange might look like below.

Assuming that my input data source has got only 2 partitions. So, stage 0 exchange should have 2 partitions. Assuming having data for 5 towers. So, each partition in this exchange might have data for tower a, tower b, c d, e . This is how exchange looks. Stage 0 will read data from source table and make it available to the exchange for the second stage to read. I have only 2 partitions in my input source. So, exchange shows only 2 partitions. Now shuffle/sort operation will read these partitions , sort them by tower location and bring them to the input exchange of stage 1. Final Result will look like this.

Let's assume configured spark.sql.shuffle.partitions = 10 . Default value for this configuration is 200. But we don't have 200 unique towers. I am running query to group by towers. So reduced number of shuffle partitions to 10. So, shuffle/sort will create 10 partitions in the exchange. Even if we have only 5 unique values, shuffle/sort will create 10 partitions. Five of them will have data and remaining five will be blank partitions. we need 10 tasks for this case because we have 10 partitions in the exchange. And that is the problem. Five partitions are empty but spark will still trigger 10 tasks. Empty partition task will do nothing and finish in milliseconds.But spark scheduler needs to spend time for scheduling and monitoring these useless tasks. Overhead is small but we do have some unnecessary overhead here. we improved situation by reducing shuffle partitions to 10 from default value of 200. But that's not an easy thing to do. Because we cannot keep changing shuffle partitions for every query.Even if we want to do that, we do not know how many unique values my SQL will fetch. Number of unique keys is dynamic and depends on data set. How am I supposed to know how many shuffle partitions do I need? It's impossible for developers to know in advance. We also have another problem here, some partitions are big, others are small as shown in above screenshot. They are not proportionate. So, task working on partition1 will take a long time while task doing partition2 will finish very quickly and that's not good because stage is not complete until all the tasks of a stage are complete. Three tasks which are processing for 2 , 3 and 4 will finish quickly but we still need to wait for partition 1 and partition 5. That's the wastage of CPU resources. Situation becomes worse when one partition becomes excessively long.

Above query resulted in 5 disproportionate shuffle partitions. after reducing shuffle partitions to 10 still result is not good.

How do we decide number of shuffle partitions for my spark job?

Spark 3.0 offers Adaptive Query Execution to solve this problem. We need to enable AQE and AQE will take care of setting the number of our shuffle partitions.

How it happens?

Our input data is already loaded in Stage 0 in write exchange. Now spark will start shuffle/sort. So it can compute statistics on this data and find out some details such as following.

```
Number of unique tower_location = 5
Number of records in tower_locations?
    tower_a = 120K
    tower_b = 40K
    tower_c = 70K
    tower_d = 50K
    tower_e = 110K
```

And this is called **dynamically computing the statistics on our data during shuffle/sort**. Such dynamic statistics are accurate and most up to date. When spark knows enough information about our data, it will dynamically adjust number of shuffle partitions for the next stage. For example, in this case, spark might dynamically set the shuffle partitions to 4 and result of that setting looks like below.

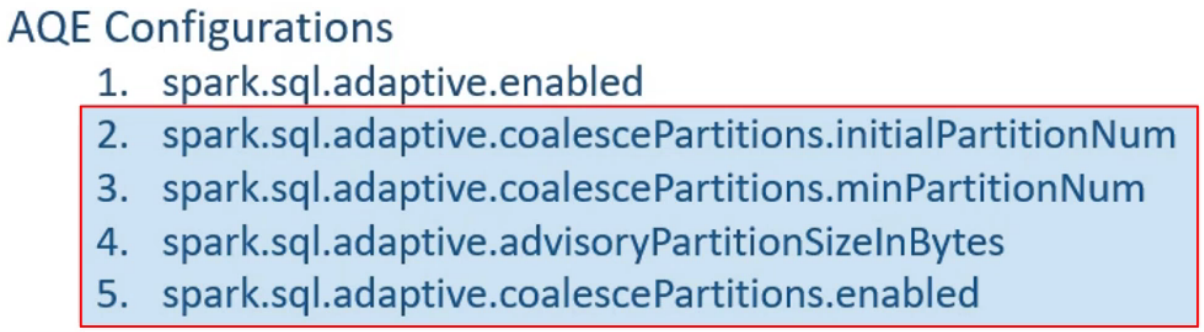
We have 4 shuffle partitions for stage 1. Those 5 empty partitions are gone. Spark also merged 2 small partitions to create one larger partition. So instead of having 5 disproportionate partitions, we have 4 partitions and these are little more proportionate as shown above. Now we need only 4 tasks to run this stage 1 because we have 4 partitions. Task3 working on partition3 will finish quickly but other three tasks will take almost equal time. So, we saved one CPU slot (partition2 and partition 4 merged) and we also eliminated useless empty tasks.

Summary: Spark shuffle/sort has critical impact on spark query performance. One fundamental tuning property of shuffle operation is number of output partitions. we can set this number using spark.sql.shuffle.partitions. It is impossible to know the best number to keep for spark.sql.shuffle.partitions because it depends on our data size and other factors. We have 2 problems here, if we configure a small number of partitions , then data size of each partition may be very large. A large shuffle partition may cause 2 types of problems. Your task may need a lot of memory to process the partition and it might slow down juggling data between memory and disk. In

the worst case, we might see an OOM exception. What if we have too many partitions? data size of each partition may be small, and there will be lot of small network data fetch to read shuffle blocks. Our query performs slowly because of the inefficient network I/O. A large number of partitions will also result in many tasks and put more burden on spark task scheduler. To solve this problem, we can set a relatively large number of shuffle partitions at the beginning and enable AQE. **AQE feature of Apache spark will compute shuffle file statistics at run time and perform 2 things. 1) To Determine the best shuffle partition number for us and set it for the next stage. 2) Combine/coalesce smaller partitions into bigger partitions to achieve even data distribution amongst the task.**

We can enable AQE using below configuration.

spark.sql.adaptive.enabled → Default value of this configuration is false. This configuration is kind of master configuration to enable/disable AQE feature. If we disable it, other 4 configurations don't take any effect. But if we enable AQE, we can tune it further using other 4 configurations.



We have 4 additional configurations to tune AQE behavior.

spark.sql.adaptive.coalescePartitions.initialPartitionNum → 1st configuration sets initial number of shuffle partitions . AQE starts with our specified value. This configuration works as max number of shuffle partitions. Spark AQE cannot set number larger than this. This configuration doesn't have a default value. So, if we donot set this configuration, spark will set it equal to spark.sql.shuffle.partitions .

spark.sql.adaptive.coalescePartitions.minPartitionNum → This configuration defines minimum number of shuffle partitions after coalescing or combining multiple partitions. we don't have default value for this configuration also. So, if we donot set this configuration,spark will set it equals to spark.default.parallelism

spark.sql.adaptive.advisoryPartitionSizeInBytes → Default value for this configuration is 64MB and it works as an advisory size of the shuffle partition during adaptive optimization. So, this configuration takes effect when spark coalesces small shuffle partitions or splits skewed shuffle partition. AQE will use this number for determining size of partitions and combine them accordingly.

spark.sql.adaptive.coalescePartitions.enabled → The default value for this configuration is true. If we set this value to false, spark AQE will not combine or coalesce smaller partitions.

SPARK ADAPTIVE QUERY EXECUTION-DYNAMICALLY SWITCHING JOIN STRATEGIES

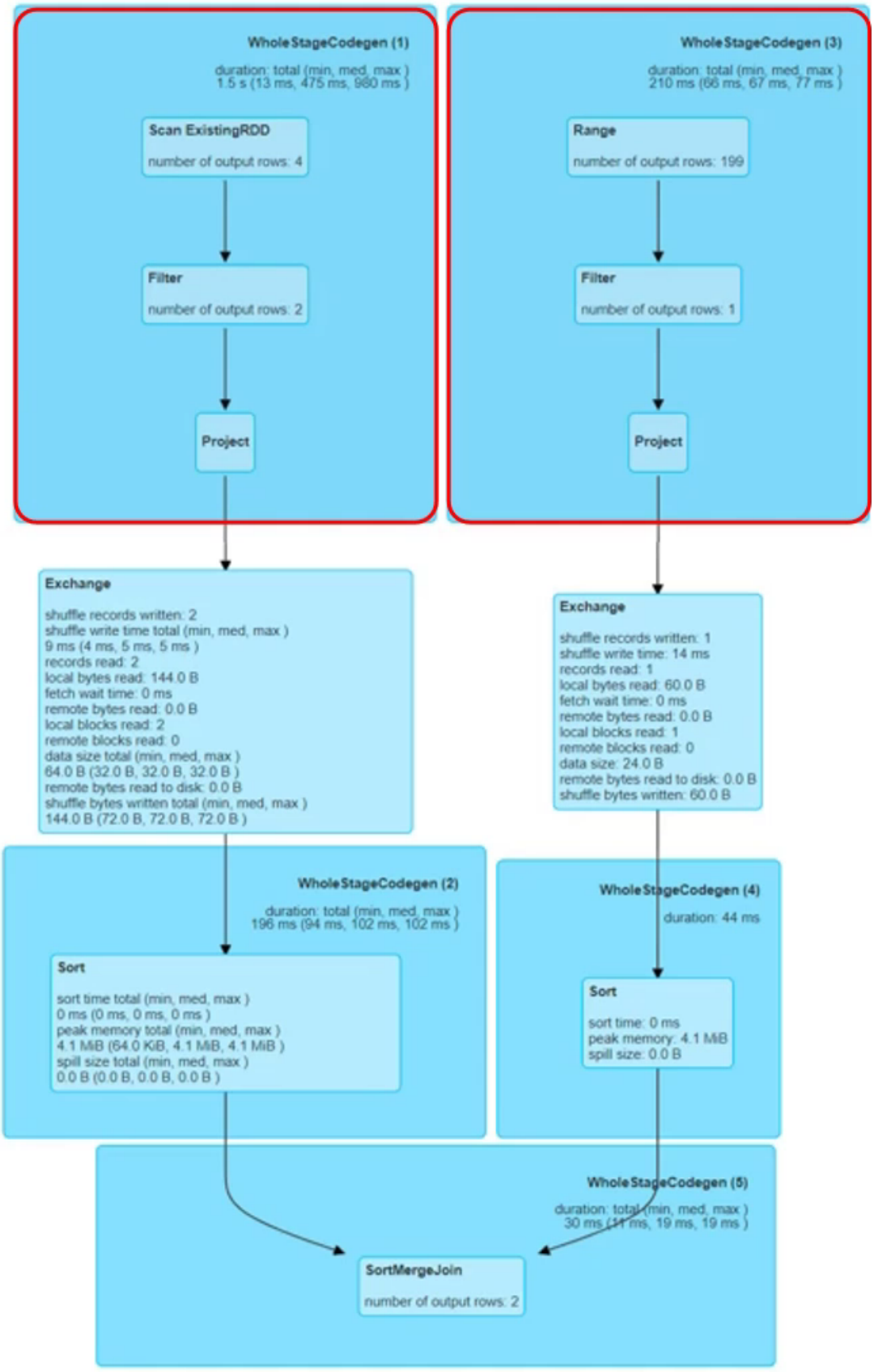
Why do we need dynamic join optimization?

Let's assume we have 2 large tables. and we are joining these 2 tables using below query. Below both queries are same only, one in data frame and other via SQL.

```
SELECT *
FROM large_tbl_1
JOIN large_tbl_2 ON large_tbl_1.key = large_tbl_2.key
WHERE large_tbl_2.value = 'some value'
```

```
df1.join(df2, df1.key == df2.key, "inner")
    .filter("value=='some value'")
```

Both of tables are large tables, so we are expecting a sort merge join to takes place. We ran our job and checked execution plan as shown below.


WholeStageCodegen (1)
duration: total (min, med, max)
1.5 s (13 ms, 475 ms, 980 ms)
Scan ExistingRDD
number of output rows: 4
Filter
number of output rows: 2
Project
WholeStageCodegen (3)
duration: total (min, med, max)
210 ms (66 ms, 67 ms, 77 ms)
Range
number of output rows: 199
Filter
number of output rows: 1
Project
Exchange
shuffle records written: 2
shuffle write time total (min, med, max)
9 ms (4 ms, 5 ms, 5 ms)
records read: 2
local bytes read: 144.0 B
fetch wait time: 0 ms
remote bytes read: 0.0 B
local blocks read: 2
remote blocks read: 0
data size total (min, med, max)
64.0 B (32.0 B, 32.0 B, 32.0 B)
remote bytes read to disk: 0.0 B
shuffle bytes written total (min, med, max)
144.0 B (72.0 B, 72.0 B, 72.0 B)
Exchange
shuffle records written: 1
shuffle write time: 14 ms
records read: 1
local bytes read: 60.0 B
fetch wait time: 0 ms
remote bytes read: 0.0 B
local blocks read: 1
remote blocks read: 0
data size: 24.0 B
remote bytes read to disk: 0.0 B
shuffle bytes written: 60.0 B
WholeStageCodegen (2)
duration: total (min, med, max)
196 ms (94 ms, 102 ms, 102 ms)
Sort
sort time total (min, med, max)
0 ms (0 ms, 0 ms, 0 ms)
peak memory total (min, med, max)
4.1 MiB (64.0 KiB, 4.1 MiB, 4.1 MiB)
spill size total (min, med, max)
0.0 B (0.0 B, 0.0 B, 0.0 B)
WholeStageCodegen (4)
duration: 44 ms
Sort
sort time: 0 ms
peak memory: 4.1 MiB
spill size: 0.0 B
WholeStageCodegen (5)
duration: total (min, med, max)
30 ms (11 ms, 19 ms, 19 ms)
SortMergeJoin
number of output rows: 2

We are reading 2 tables and we have 2 stages of reading those tables. Both these stages send data to exchange and everything after exchange is part of stage 3. So, my 3rd stage collects data from the exchange , performs sorting operation on the data and finally joins them. That's what happen in sort-merge join operation.

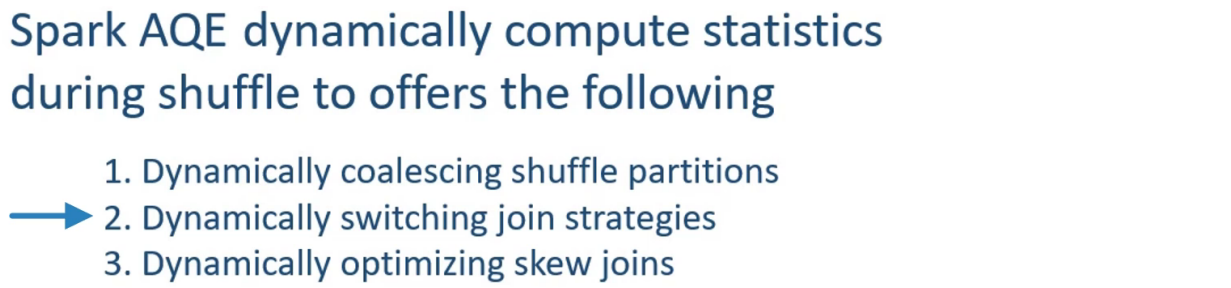
We are assuming both tables are large and also in query we applied filter on large_tbl_2 . large_tbl_2 is a 100GB table. But what is the effect of applying a filter ? we did some investigation and realized we selected only 7MB of data from large_tbl_2 . our large_tbl_1 is 100GB and we are selecting all rows from this table. our large_tbl_2 is also 100GB but we are selecting only 7MB from that table.

If we know this information already, will we apply for sort-merge join? No. because I want to use broadcast hash join here because one of my table is small enough . so we can see broadcast hash join and avoid shuffle/sort operation. But in this case why broadcast join is not happening ? because spark will not apply broadcast hash join if broadcast join threshold is broken. So lets check below configuration.

spark.sql.autoBroadcastJoinThreshold = 10MB

10 MB is default value, we haven't changed it.

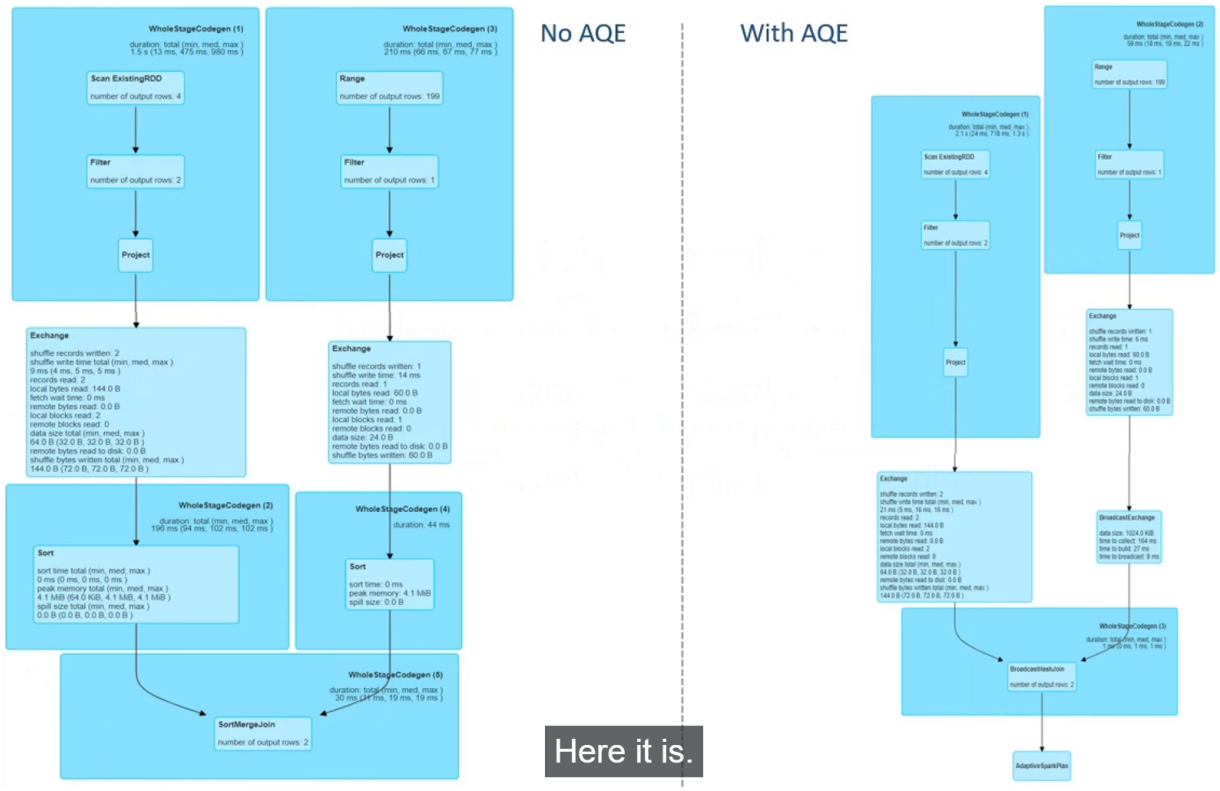
We are selecting 7MB from larger table, that is below threshold limit, but spark not applying broadcast join, why? Spark execution plan is created before spark starts the job execution. Spark doesn't know size of the table,so it applies sort-merge join. If we compute statistics on table, it may/may not apply broadcast join. Spark will not apply broadcast if we donot have a column histogram for the filter column. It can't apply broadcast join if our statistics are outdated. So, one solution is to analyze our spark tables and keep our table and column statistics up to date. Another solution is to enable AQE. Spark AQE can help us in this situation. AQE will computes statistics on the shuffled data and use that information to do following things.



Now we enabled AQE and executed same query to check new execution plan.

```
SELECT *
FROM large_tbl_1
JOIN large_tbl_2 ON large_tbl_1.key = large_tbl_2.key
WHERE large_tbl_2.value = 'some value'
```

```
df1.join(df2, df1.key == df2.key, "inner")
    .filter("value=='some value'")
```

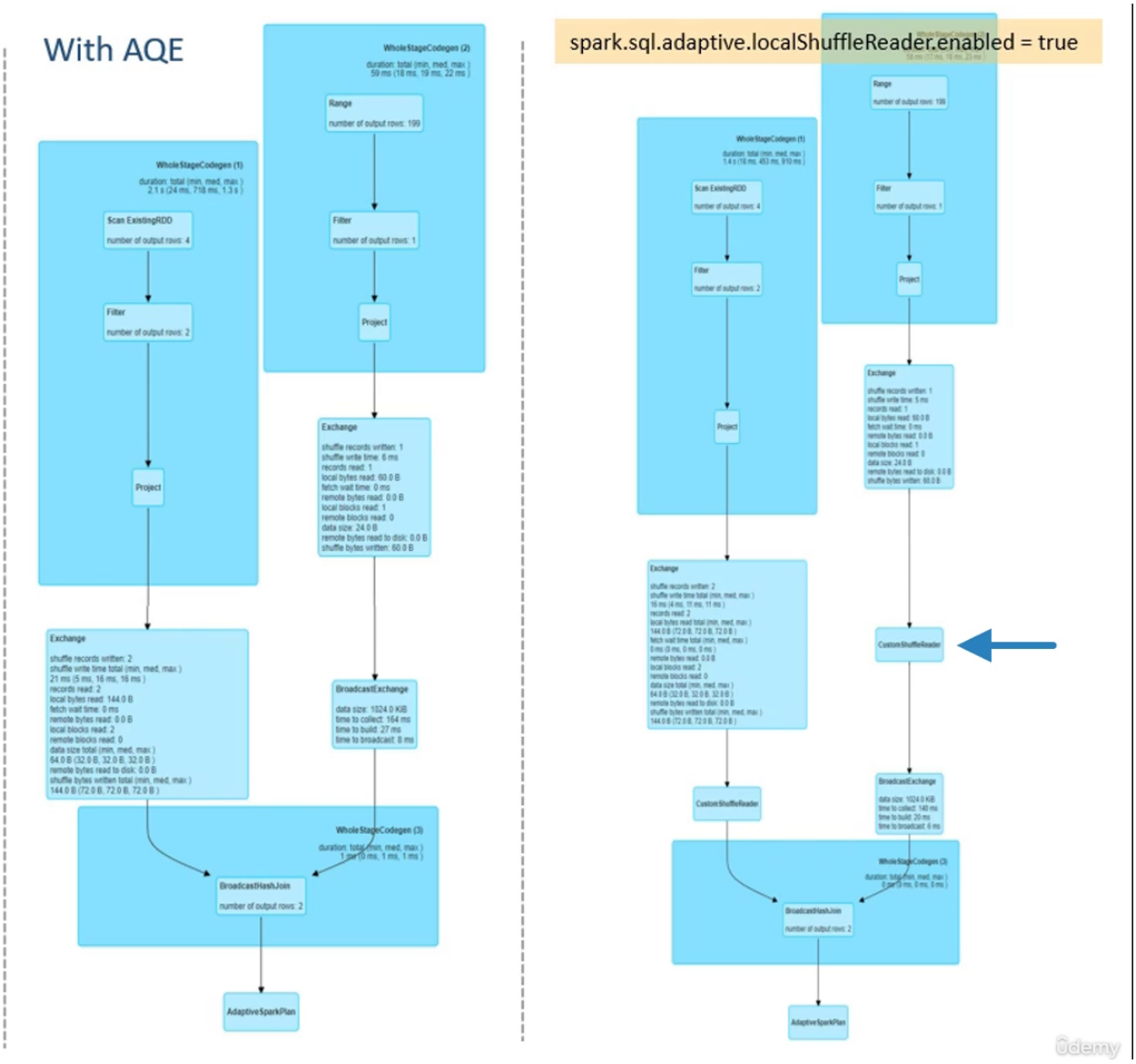


After enabling AQE also we still have 3 stages, Stage 1 and Stage 2 are for scanning the tables and sending data to exchange. But we enabled AQE. So Adaptive query will compute statistics on exchange data. Statistics tells that data size of large_tbl_2 is small enough to apply broad cast join. So AQE will dynamically change execution plan and apply broadcast hash join. We can see that in new query plan(enabling AQE). Unfortunately, we still have shuffle (in exchange box as shown above), but we saved sort operation. We couldn't save the shuffle operation, but we still see the exchange in query plan. But the sort operation is gone. AQE cannot avoid shuffle. Why? Because AQE computes statistics during the shuffle. So, shuffle operation will be there. But AQE will dynamically change the plan and apply broadcast hash join to save the expensive sort operation. But we still have some problem here, shuffle operation is already complete. We already distributed data from stage 1 and 2 to stage 3. But if we apply broadcast join now, are we going to broadcast table once again? Yes.

AQE also gives us another configuration.

spark.sql.adaptive.localShuffleReader.enabled = true → Default value for this configuration is true.

If this setting for this configuration is true, our optimized execution plan looks like this.



Both plans looks same. But we will see custom shuffle Reader/local shuffle reader. **This custom shuffle reader is specifically designed to further optimize the AQE broadcast join by reducing the network traffic. So if we are using AQE, donot disable local shuffle reader. It is by default enabled to true.**

This is how AQE dynamically switches join strategy from sort-merge join to broad cast hash join.

Summary:

Spark supports many join strategies. However, broadcast hash join is usually most performant. But we can apply broadcast join if one side of join can fit well in the memory. And for this reason, spark

plans a broadcast hash join if estimated size of join table is smaller than broadcast size threshold. But estimating table size is problematic in 2 scenarios. 1) We applied a highly selective filter on the table , in this case an up-to-date table and column statistics might help. 2) If the join table is dynamically created after series of complex operations in this case, statistics are not even applicable. Spark AQE replans join strategy at run time based on the most accurate join relation size to solve this problem. **So AQE calculates table statistics at run time using shuffle exchange. And reoptimizes query plan converting sort-merge join to broadcast hash join.**

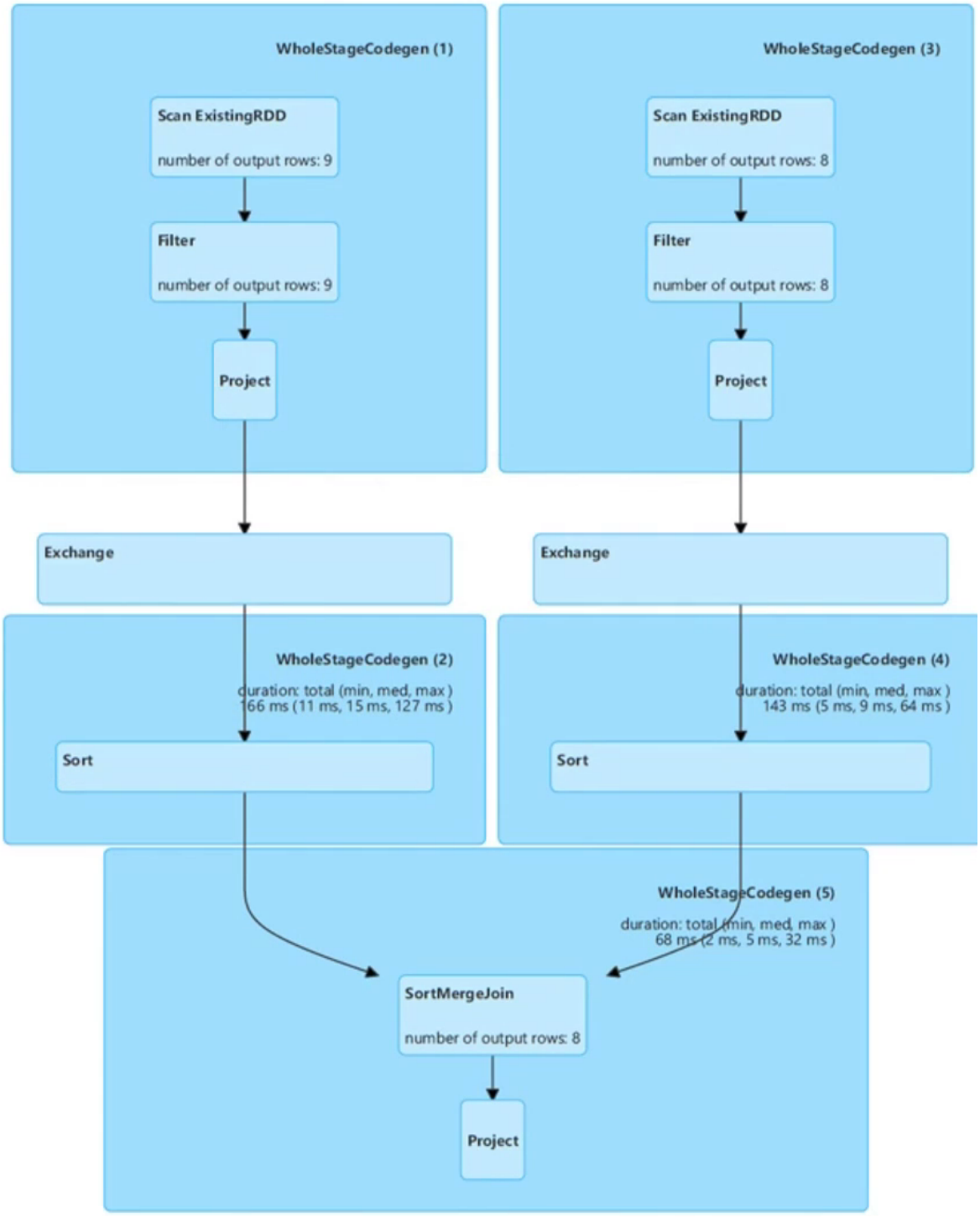
SPARK ADAPTIVE QUERY EXECUTION - HANDLING DATA SKEW IN SPARK JOINS-DYNAMICALLY OPTIMIZING SKEW JOINS

Let's assume we have 2 larger tables and we are joining these 2 tables using below query:

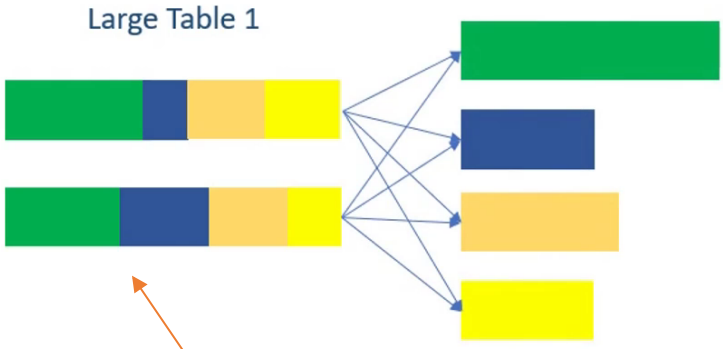
```
SELECT *
FROM large_tbl_1
        JOIN large_tbl_2
        ON large_tbl_1.key = large_tbl_2.key
```

```
df1.join(df2, df1.key == df2.key, "inner")
    .filter("value=='some value'")
```

Above both commands are same. Both tables are large tables so we are expecting a sort-merge join to takes place. We ran job and checked execution plan as shown below. We are reading table 1 and table 2 . These tables should join, so we have a shuffle operation for both tables. so that's why we see 2 exchanges in below screenshot. The first exchange partitions the data by join key for the first table and the second exchange partitions the data by join key for second table.

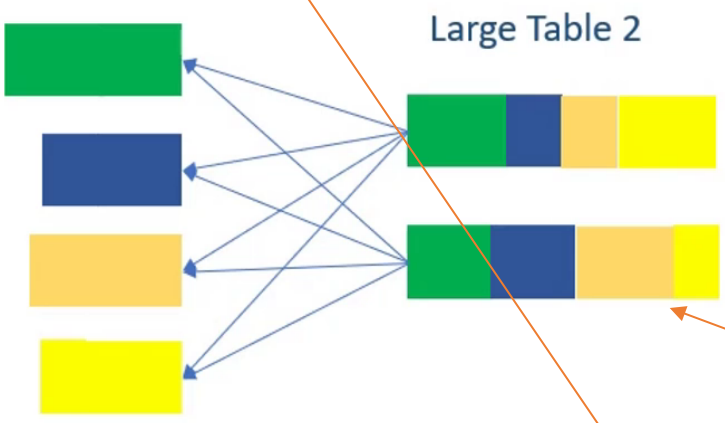


Let's assume that my first table had 2 partitions as shown below in left hand side . Each color represents data for one unique join key. So, we read these 2 initial partitions and then we shuffled them. The result of shuffle looks like below as in RHS.



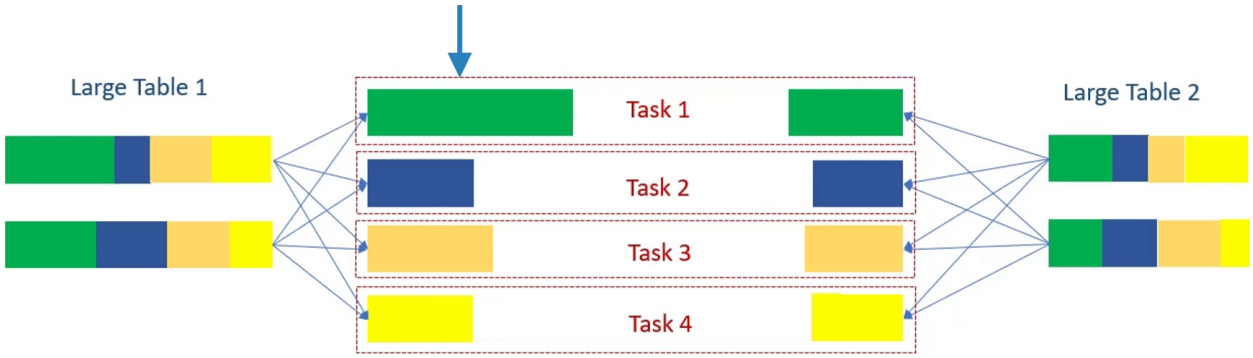
Primary purpose of shuffle is to repartition data by the key.

Second table also go through same process.



So, partitions for the first table are coming from left side, and second table comes from right side.

Spark appropriately partitioned data by the key. Now all we need to do is sort each partition by the key and merge records from both sides. I have 4 partitions to join so we will need 4 tasks. Each task will pickup one color, sort the data and merge it to complete join operation.



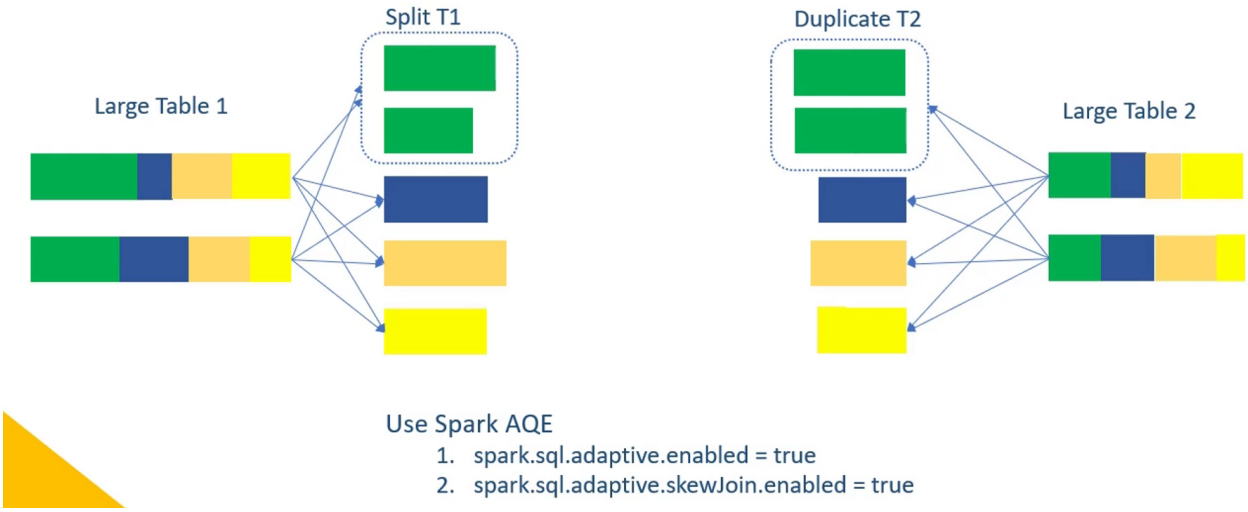
Green color partition on the left side is quite large as shown above. Green partition is skewed . Blue orange and Yellow was smaller but green one is almost double the size, that's the problem and we

need more memory to sort and merge green partition. We planned 4GB RAM for each task, and that should be sufficient for other tasks. But the task working with green color cannot manage sort/merge operation with 4GB RAM. Can we increase spark memory ? Yes, we can increase it but its not a good solution. Because we have 2 reasons. 1) The first reason is memory wastage. We can increase memory assuming that we will have skewed partition, But we are not sure if all our joins will result in a skewed partition. What if they don't? You might have 10/15 join operations in our spark application. All other join operations work perfectly fine with 4GB task memory. However, we have 1 skewed join and one task needs extra memory. Increasing memory for 1 specific join/task is not possible. So, we will end up increasing memory for our entire application, and that's the wastage. 2) Increasing memory for skewed join is not a permanent solution, because data keeps changing. We might have a skewed partition today, which required 6GB to complete sort/merge operation. But we do not know what happens a week later. We got new data and skew is now more significant. We now need 8GB to pass through that skew. We cannot let my application fail every week or month, investigate logs, and identify that we need more memory because data is now more skewed. So whats the solution? Spark AQE offers us an excellent solution for this problem.

We can enable skew optimization using below configurations.

1) **spark.sql.adaptive.enabled = true → This configuration enables Spark AQE feature.**
2) **spark.sql.adaptive.skewJoin.enabled = true → This configuration enables skew-join optimization.**

I have 4 shuffle partitions , so we need 4 tasks to perform this sort/merge join operation. However green partition is skewed,So green task will struggle to finish and take longer to complete. In worst case, it might fail due to memory crunch. But other 3 tasks will complete quickly and normally because they are small enough. We now enabled AQE and skew-join optimization. So, Spark AQE will detect this skewed partition and it will do 2 things. Split the skewed partition on left side into two/more partitions. In our example splitting 2 is sufficient, so let's assume spark splitting it into two partitions. spark will also duplicate matching right-side partition.



Now problem solved. Now we have 5 partitions. earlier we have 4. Now we have 5 partitions so we will need 5 tasks. Data partitions for all five tasks are almost the same. So, they will consume uniform resources and finish simultaneously. That's how AQE and Skew Join optimization works.

We also have 2 more configurations to customize skew join optimization. Below 2 configurations are used to define the skew.

When do we consider a partition as a skewed partition and start splitting it ? Spark AQE assumes that partition is skewed and starts splitting when both thresholds are broken.

3) **spark.sql.adaptive.skewJoin.skewedPartitionFactor → Default value of skewed Partition Factor is 5 . So, a partition is considered skew if its size is larger than 5 times the median partition size.**
4) **spark.sql.adaptive.skewJoin.skewedPartitionThresholdInBytes → Default value of skewedPartitionThresholdInBytes is 256MB . So if partition is considered skewed, if its size in bytes is larger than this threshold.**

NOTE: Spark AQE will initiate the split if only if both thresholds are broken.

SPARK DYNAMIC PARTITION PRUNING

Dynamic Partition Pruning is a new feature available in spark 3.0 and above and this feature is enabled by default. However if we want to disable it, we can use following configuration.

spark.sql.optimizer.dynamicPartitionPruning.enabled → default value for this configuration is true in Spark 3.0

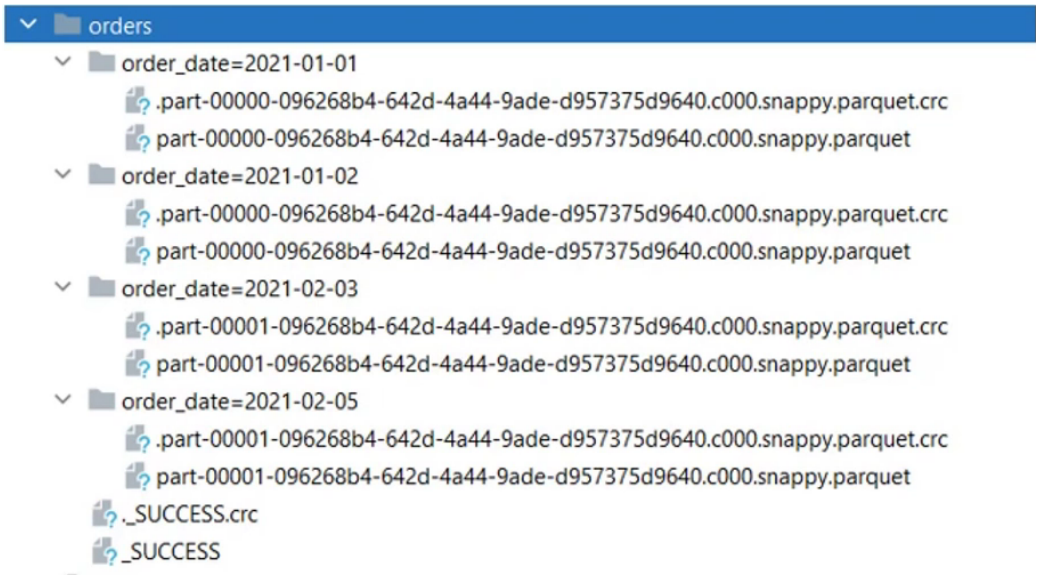
WHY DO WE NEED DYNAMIC PARTITION PRUNING?

Let's assume we have 2 tables. 1st table is an orders table and it stores all our orders. Table structure looks like below . This table is huge because it stores thousand of orders every day and millions of orders a year. So we decided to partition it on order_date.

Orders

order_id	order_date	prod_id	unit_price	qty
101	2021-01-01	PDX1	350	7
102	2021-01-02	LDX1	580	5
102	2021-01-02	PDX1	350	3
104	2021-02-03	LDX1	580	8
105	2021-02-03	PDX1	350	6
106	2021-02-05	LDX1	580	9

Below is a screenshot of how my orders data set is stored.



Why do we partition it on order_date? Because we are querying this table on order_date.

```
order_df = spark.read.parquet("orders")
summary_df = order_df
    .where("order_date=='2021-02-03'") \
    .selectExpr("sum(unit_price * qty) as total_sales")
```

The above command reading orders dataset , then applying a filter on order_date and finally computing total sales for 3rd Feb 2021. We ran this code and checked query plan as shown below.

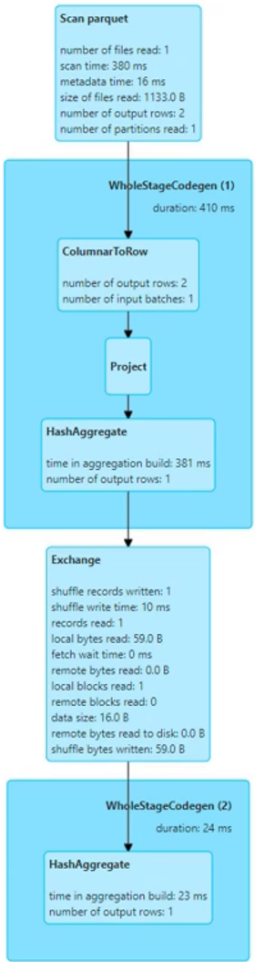
The below image shows physical plan for this query. It clearly shows that spark performed 6 steps to complete this query. The first step is to scan/read the parquet file. But spark is doing smart thing here. In Output[3] : **Partition Filters:** is there as shown below. Spark applied partition filters on the order_date column and it is reading only 1 partition. Plan also shows number of files read as 1 in logical plan(scan parquet box). So, what's happening here? In typical case, spark should read all the partitions of my parquet data set and then apply the filter. But my data set was partitioned on the order_date column. So, spark decided to read only 1 partition that belongs to 3rd Feb 2021.

Some Features of Spark Query Optimization: Predicate Push down and Partition Pruning.

Predicate Push down means spark will push down where clause filters down in the steps and apply them as early as possible. I mean spark will not try typical sequence to read data first , then filter, and finally calculate the sums. Spark will push filter condition down to the scan step and apply where clause when reading the data itself. But predicate push down doesn't help much unless our data is

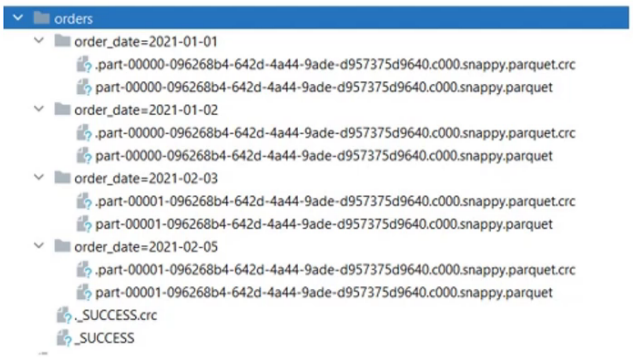
▼ Details

```
== Physical Plan ==
* HashAggregate (6)
+- Exchange (5)
   +- * HashAggregate (4)
      +- * Project (3)
         +- * ColumnarToRow (2)
            +- Scan parquet  (1)


(1) Scan parquet
Output [3]: [unit_price#2L, qty#3L, order_date#4]
Batched: true
Location: InMemoryFileIndex [file:/E:/demo/PySpark/data]
PartitionFilters: [isnotnull(order_date#4), (order_date#4 = 18661)]
ReadSchema: struct<unit_price:bigint,qty:bigint>
```



▼ Details

```
== Physical Plan ==
* HashAggregate (6)
+- Exchange (5)
   +- * HashAggregate (4)
      +- * Project (3)
         +- * ColumnarToRow (2)
            +- Scan parquet  (1)


(1) Scan parquet
Output [3]: [unit_price#2L, qty#3L, order_date#4]
Batched: true
Location: InMemoryFileIndex [file:/E:/demo/PySpark/data]
PartitionFilters: [isnotnull(order_date#4), (order_date#4 = 18661)]
ReadSchema: struct<unit_price:bigint,qty:bigint>
```



partitioned on the filter columns. Our data set was already partitioned on order_date. So, spark decided to read only 1 partition for 3rd Feb 2021. It can simply leave all other partitions and that feature known as partition pruning. Predicate push down and Partition Pruning – these 2 features will optimize our query and reduce the read volume of our data. If we are reading less amount of data, our queries run faster.

Now WHAT IS DYNAMIC PARTITION PRUNING:

Let's say we have another table , we call it as Dates table. This table known as date dimension in the data warehousing world. The above orders table is a fact table. And dates table is a dimension table. This kind of table structure is common in data warehouses. Let's assume we are running query like below. We want to calculate sum of sales for Feb 2021. Here is an equivalent data frame expression. SQL and data frame expressions are same.

```
SELECT year, month, sum(unit_price * qty) as total_sales
FROM orders JOIN dates ON order_date == full_date
GROUP BY year, month
WHERE year=='2021' and month=='02'
```

```
join_expr = order_df.order_date == date_df.full_date

order_df
    .join(date_df, join_expr, "inner") \
    .filter("year==2021 and month==2") \
    .groupBy("year", "month") \
    .agg(f.sum(f.expr("unit_price * qty"))
    .alias("total_sales"))
```

Orders - Fact

order_id	order_date	prod_id	unit_price	qty
101	2021-01-01	PDX1	350	7
102	2021-01-02	LDX1	580	5
102	2021-01-02	PDX1	350	3
104	2021-02-03	LDX1	580	8
105	2021-02-03	PDX1	350	6
106	2021-02-05	LDX1	580	9

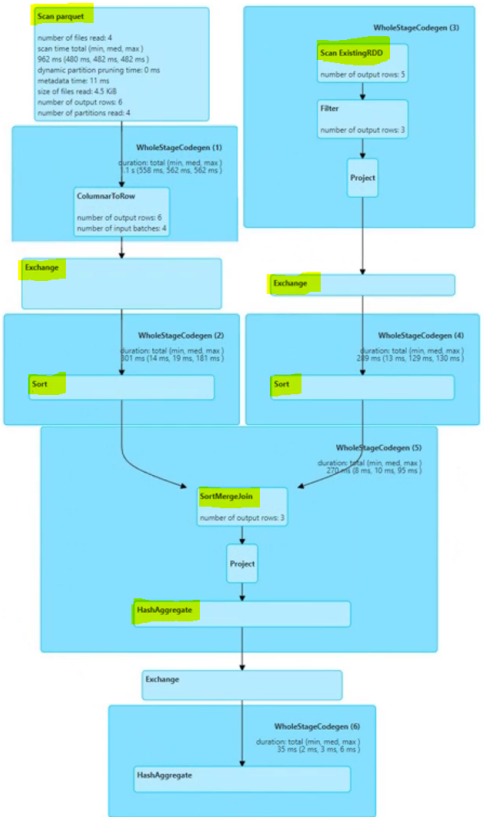
Dates - Dimension

full_date	year	month	day
2021-01-01	2021	01	01
2021-01-02	2021	01	02
2021-02-03	2021	02	03
2021-02-04	2021	02	04
2021-02-05	2021	02	05

We can execute above code on an older version of spark where dynamic partition pruning is not available or we can disable dynamic partition pruning in newer version and try this SQL.

We will get execution plan as below . This is a typical sort/merge join execution plan. We are reading parquet files and also reading other dataset. Both are going for a shuffle operation and hence we can see these two exchanges as shown in below screenshot. Finally spark will sort, then merge and finally aggregate it. My parquet files are partitioned on the order date column but that partitioning does not benefit us. If we see scan parquet step in the diagram , we can see number of files read is 4. Physical plan shows partition filter step , but it is not pruning any partitions because we do not have any filter condition on the order date. In above SQL query, we are filtering for year and month columns on dates table i.e.; we want to read all the partitions for Feb 2021. But spark is reading other months partitions also . The best case is to read only Feb 2021 partitions and leave all other partitions. However, Spark is not applying partition pruning here, because filter conditions are on the dates

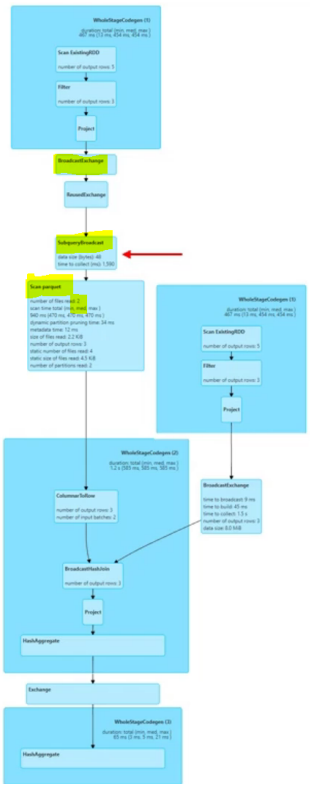
table. They are not on the orders table. How to improve it? We need to do 2 things, Enable Dynamic Partition Pruning feature and apply broadcast on the dimension table.



Dynamic Partition Pruning is enabled by default in spark 3.0 and above. So, we do not need to enable it because we are using spark 3.0 . But we must apply broadcast to my dimension table. So, we modified code and it will look like this.

```
join_expr = order_df.order_date == date_df.full_date

order_df
    .join(f.broadcast(date_df), join_expr, "inner") \
    .filter("year==2021 and month==2") \
    .groupBy("year", "month") \
    .agg(f.sum(f.expr("unit_price * qty")) 
    .alias("total_sales"))
```

This code is same as earlier, But now we have applied broadcast() to date_df . we ran this code again and below is the new execution plan. Here date_df going for a broadcast exchange. But then spark creates a subquery from the broadcasted date_df and sending it as an input to parquet scan and we can see number of files read in scan parquet box. We are now reading only 2 files. So, what happened here is spark applied partition pruning and now it is reading only February 2021 partitions. If we look at the physical plan and check out the scan parquet details, we will see dynamic pruning expression. Spark Dynamic partition pruning can take a filter condition from our dimension table and inject it into our fact table as subquery. Once subquery is injected into our fact table, spark can apply partition pruning on our fact table.



Details

```
== Physical Plan ==
* HashAggregate (11)
+- Exchange (10)
   +- * HashAggregate (9)
      +- * Project (8)
         +- * BroadcastHashJoin Inner BuildRight (7)
            :- * ColumnarToRow (2)
            :  +- Scan parquet  (1)
            +- BroadcastExchange (6)
               +- * Project (5)
                  +- * Filter (4)
                     +- * Scan ExistingRDD (3)


(1) Scan parquet
Output [3]: [unit_price#2L, qty#3L, order_date#4]
Batched: true
Location: InMemoryFileIndex [file:/E:/demo/PySpark/orders]
PartitionFilters: [isnotnull(order_date#4), dynamicpruningexpression(order_date#4 IN dynamicpruning#58)]
ReadSchema: struct<unit_price:bigint,qty:bigint>
```

It works like magic. But using this feature is not straight forward. We must understand the below things.

1. Works for fact and dimension-like tables
2. Fact table must be partitioned
3. Broadcast dimension table

We must have fact and dimension like set-up. When I say fact and dimensions, I mean one large table and another small table. Our large table or fact table must be partitioned so spark can try partition pruning. We must broadcast our smaller table or dimension table. If it is smaller than 10MB, then

spark should automatically broadcast it. However, we must make sure that our dimension table is broadcasted. If we meet all three conditions, spark will most likely apply dynamic partition pruning to optimize our queries.

DATA CACHING IN SPARK

Spark Memory Pool is broken down to 2 sub-pools. Storage Memory and Executor Memory. Spark will use executor pool to perform data frame computations. And we can use storage pool for caching data frames.

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?

Answer to all these above questions are essential and critical for any spark developer.

HOW DO WE CACHE DATA FRAME IN THE EXECUTOR STORAGE POOL ?

We have 2 methods for doing this cache() and persist() method.

WHAT IS DIFFERENCE BETWEEN CACHE() AND PERSIST() ?

Both of these methods will cache our data frame in the executor storage pool. At a high level, they are same because they do same thing – cache the data frame. But there is a slight difference. cache() method doesnot take any argument. However, persist() method takes an optional argument for the storage level. So, signature for persist method look like below.

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

```
persist([StorageLevel.storageLevel])
```

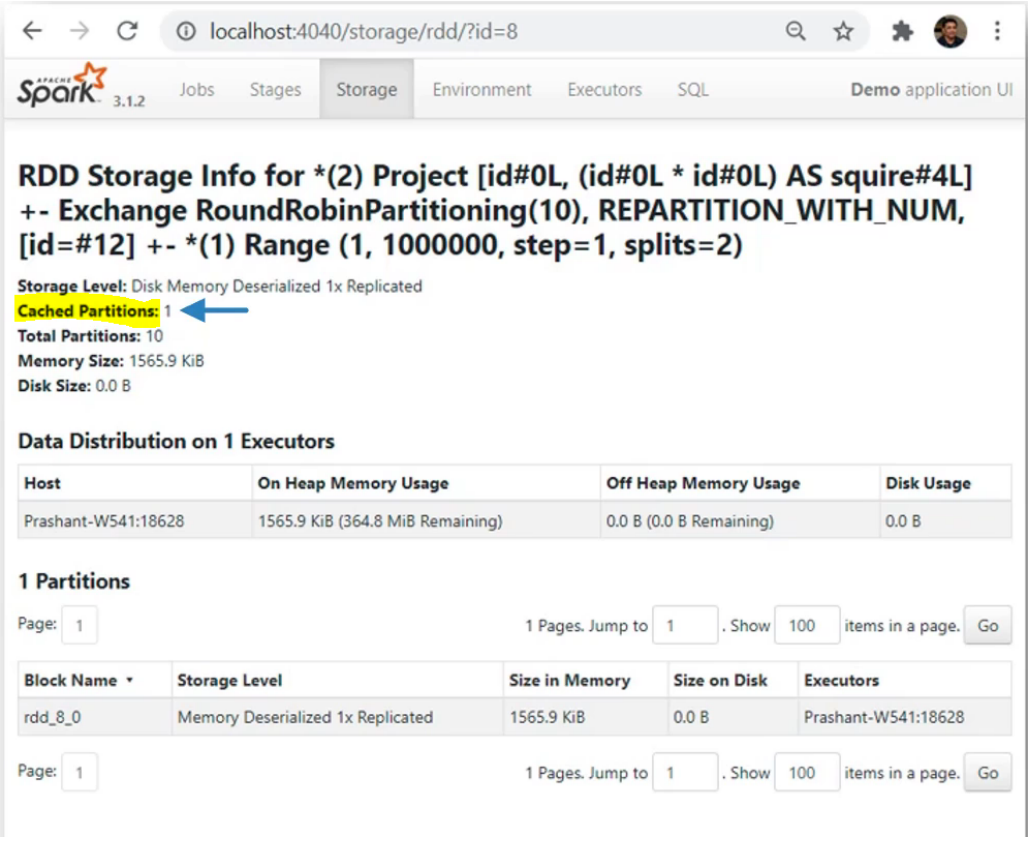
We have an alternative way of persist command as shown above. Both of these alternatives are to specify storage level. Cache method doesnot take any argument and it will cache our data frame

using default MEMORY_AND_DISK storage level. However, persist() method is more flexible. persist() method will also cache data frame , but it allows us to configure storage level.

BUT WHAT IS STORAGE LEVEL IN PERSIST COMMAND ? lets understand with simple example.

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

df.take(10)
```

Here we are creating data frame of 1 million rows and repartitioning it to 10 uniform partitions. We repartitioned it just to see how partitioned data frame is cached in our spark memory. If we are loading data frame from parquet/other types of files, our data frame is most likely to have multiple partitions. But we are creating a data frame at run time, so we have to repartition it to show how multiple partitions behave when we cache it. Then we are calling cache() method to cache my data frame in spark storage memory. Finally executing take(10) action. cache() method is a lazy transformation so spark will not do anything until we execute an action. That is why we are executing take(10) action. take(10) action will load one partition to spark memory and give us 10 records. We ran above code and checked the storage tab of my spark UI. The below is the screenshot.



We have 1 partition in the cache as shown in above screenshot. This above row shows details of my cached partition.

Summary about above screenshot :

We created a data frame of 10 partitions. Then we cached it. But spark is showing only 1 partition in the cache because we used take(10) action. take(10) action will bring only one partition in the memory and return one record from loaded partition. Since we brought only one partition to memory, spark will cache only 1 partition. Spark would cache the whole partition. We are taking only 10 records from loaded partition, but spark will cache the entire partition which means that spark will always cache entire partition. If you have a memory to cache 7.5 partitions, spark will cache only 7 partitions because 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. Default storage level for the cache() method is Memory Deserialized 1X replicated. We can also see size of cached partition in above screenshot (Size in Memory column).

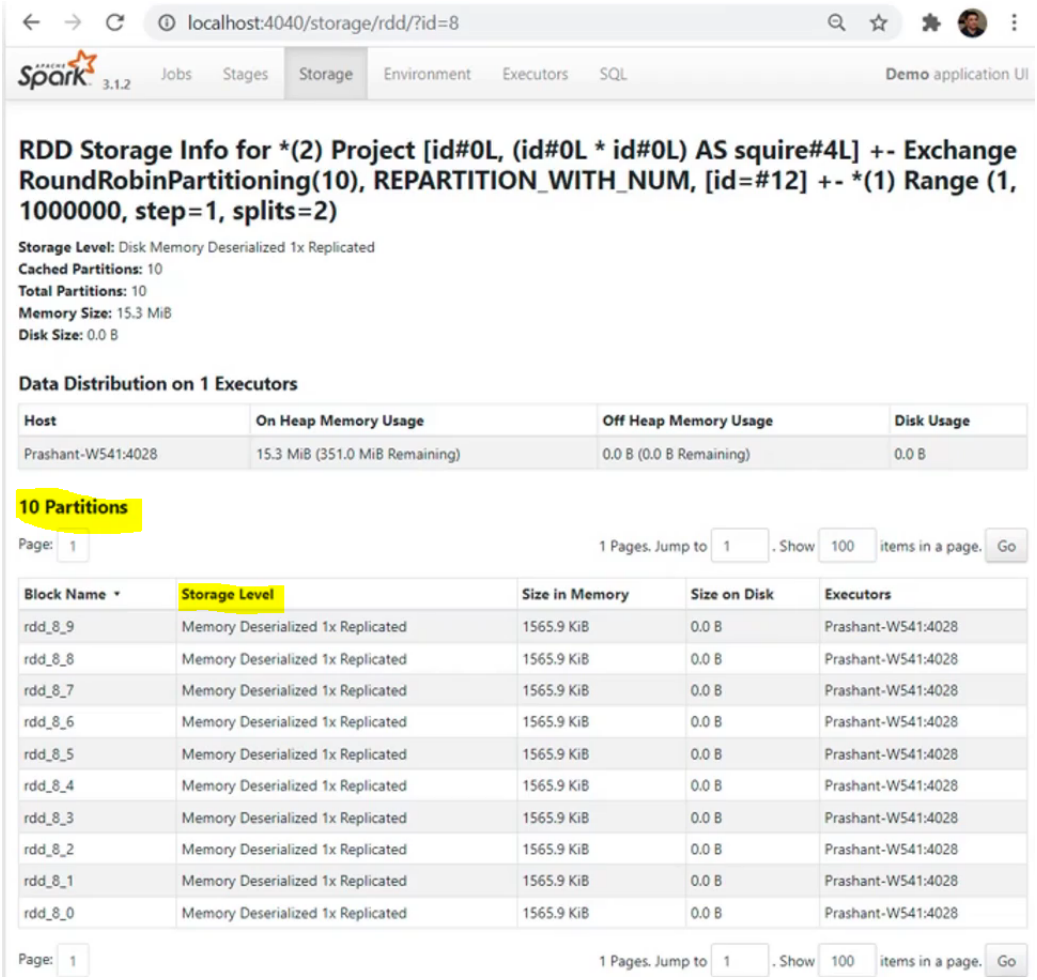
Now we changed take(10) method and replaced it with count() action. count() action will bring all the partitions in the memory, compute the count, and return it.

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

df.count(    )
```

Now we ran the code and check the storage tab once again. Screenshot shown below, Now we can see all 10 partitions shown below. Now we can see all 10 partitions are cached() because count() action forced all the partitions to be loaded in the executor memory. We have enough memory to cache all 10 partitions , so spark is caching all the 10 partitions.

Storage level of all the cached blocks is still the same as Memory Deserialized 1X replicated.

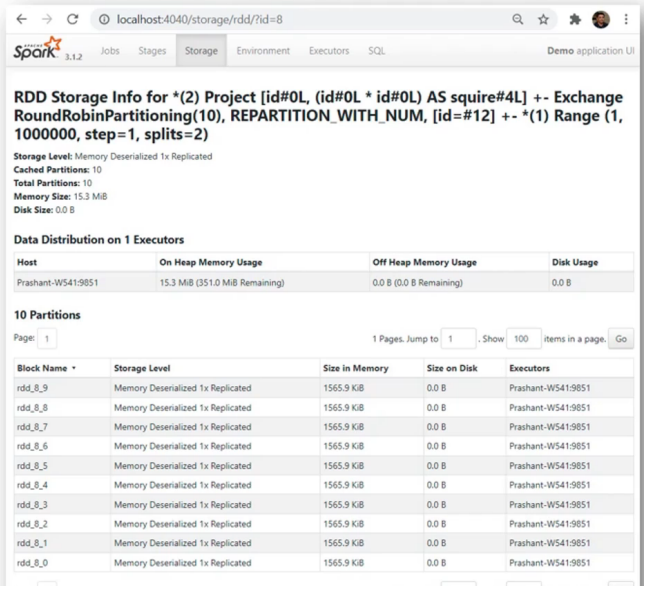
Spark cache() as well as persist() methods are lazy transformations. So, they do not cache() anything until an action is executed. cache() and persist() methods are smart enough to cache only what we access. So, if we are accessing only 3 or 5 partitions, spark will cache only those partitions. spark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



Now let's use persist() method and change the storage level.

We are using persist method instead of cache. Below is the screenshot of spark UI.

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



cache() method and persist() method work in the same way. persist() method allow us to customize storage level, but rest is same as cache() method. In this example, we configured storage level to the same as cache() storage level. So, we don't see any difference. But now let us change storage level and understand storage characteristics. Here is the format for StorageLevel method.

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

Here we are saying use Memory to cache spark data frame partitions. **useDisk** True means extending cache to disk. If partition doesnot fit into spark memory, spark will cache it in the local disk of the executor. I can also extend it to off heap memory(useOffHeap) for caching my data frame partitions. However we will add some off-heap memory to spark. If we haven't added off-heap memory for our spark application,setting this parameter(useOffHeap) will not have any effect. Now we go to **deserialized.** 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 we cache data in memory, we can cache it in serialized format or deserialized format. deserialized format takes a little extra space and serialized format is compact. So, we can save some memory using serialized format. But when spark wants to use that data, it must deserialize it. So, we will spend some extra CPU overhead which means that we can cache our data in serialized format and save some memory but incur CPU costs when spark access cached data. Alternatively, we can cache it deserialized and save our CPU overhead. The choice is ours but recommendation is to keep it deserialized and save our CPU. And this configuration only applies to memory. Our disk-based cache is always serialized.

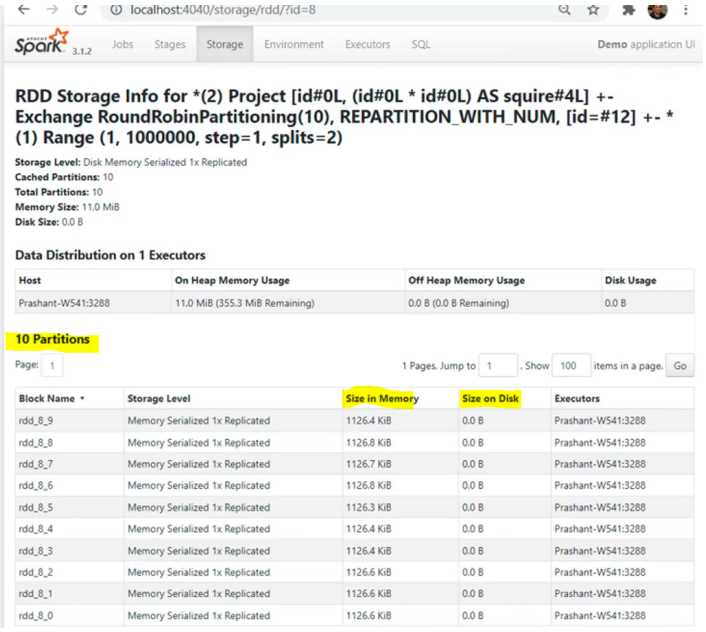
The below is new code example

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

This time, we are setting 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. We are trying to cache our data in serialized format. So, setting deserialized to false. The last one is the replication factor. I am setting it to 1 because I do not want to cache multiple copies of my data. If we set this value to 3, spark will cache 3 copies of our dataframe partitions. All those 3 copies will be cached on 3 different executors but caching multiple copies is

wastage of memory. We can cache multiple copies if we 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(    )
```



We still see everything cached in memory(size in memory column) because we 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 we do not have enough memory so size on disk is 0 . This time we store it as serialized. So, we can see the size of each partition is smaller than earlier. When we store it as deserialized, it took more memory space but serialized storage takes a little lesser.

Summary: persist method allows us to customize our cache storage level and cache it.

- We can cache our data in memory only, or we can cache it in disk only. We can also configure it to store both in memory and disk.

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

- If we have some off-heap memory , we can also expand our cache memory to off-heap storage.

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

- So, assume we enabled all three storage levels, then spark will cache our data in memory. That's the first preference.

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

If we need more storage, spark will use off-heap memory.

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

If we still want to cache more data, spark will us local disk.

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

- We can keep our cache in serialized or in deserialized formats. But this deserialization format only applicable for memory. Disk and off-heap do not support deserialized formats. They always store data in serialized formats.

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

- The last option is to increase replication factor of our cached blocks. Increasing replication can give better data locality to spark task scheduler and make our application run faster. However, it will cost us a lot of memory.

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

We used above StorageLevel function just to see persist method. But we can also use some predefined constants. The below is an example.

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

In this example, we are using a predefined storage level constant. Constant make our code more readable , but we can also use StorageLevel() function.

The below is an indicative list of available constants.

1. DISK_ONLY
2. MEMORY_ONLY
3. MEMORY_AND_DISK
4. MEMORY_ONLY_SER
5. MEMORY_AND_DISK_SER
6. OFF_HEAP
7. MEMORY_ONLY_2
8. MEMORY_ONLY_3

From above screenshot, The numbers 2 and 3 in these constants represents replication factor.

HOW TO UNCACHE?

We can remove our data frame from the cache that we previously cached using unpersist() method. We do not have any method called uncache() but unpersist() will do the job.

WHEN SHOULD WE CACHE AND WHEN SHOULD WE AVOID CACHING OUR DATA FRAMES?

When we want to access large data frames multiple times across spark actions, we should consider caching our data frames. But we must make sure to know our memory requirements and configure our memory accordingly. Do not cache our data frames when significant portions of them don't fit in the memory. If we are not reusing our data frames frequently or data frame is too small , do not cache them.

REPARTITION AND COALESCE

Spark offers 2 repartition methods.

Repartition your Dataframe

1. repartition(numPartitions, *cols)
   - Hash based partitioning
2. repartitionByRange(numPartitions, *cols)
   - Range of values based partitioning
   - Data sampling to determine partition ranges

Both methods are almost same except for 1 difference. repartition() method will use hash function to determine the target partition of our row. However, repartitionByRange() will partition our data using range of values. For example, 0-10 goes in the first partition , then 10-20 goes in the second partition and so on. That's the only difference. otherwise, both functions work in the same way.

The repartitionByRange() method internally uses data sampling to estimate the partition ranges. So, the output of repartitionByRange() may not be consistent, which mean if we execute same code twice, we may see different partition ranges. And that should be perfectly fine in many cases. repartition() method takes 2 optional arguments. Number of partitions and list of columns. One of these 2 arguments is mandatory. So all the below calls are valid only.

repartition(numPartitions, *cols)

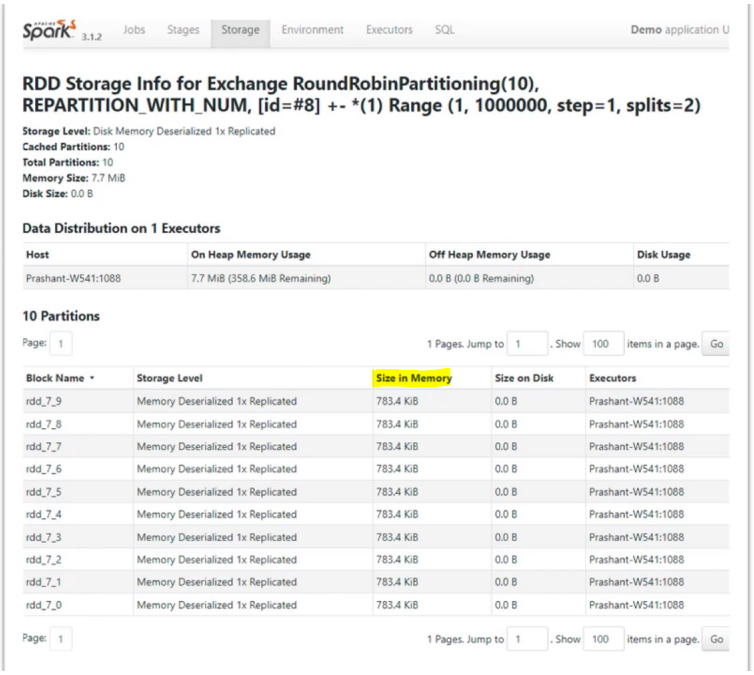
Examples

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

Similarly we can also assume repartitionByRange() also has same structure.

1 example explanation: repartition(10) will create ten new partitions of our dataframe. How? Repartition is a wide dependency transformation. So, it will cause shuffle/sort of our data frame and final output will have 10 new partitions. And all those 10 partitions will be uniform in size. Here is an example code and a screenshot of spark UI to explain it.

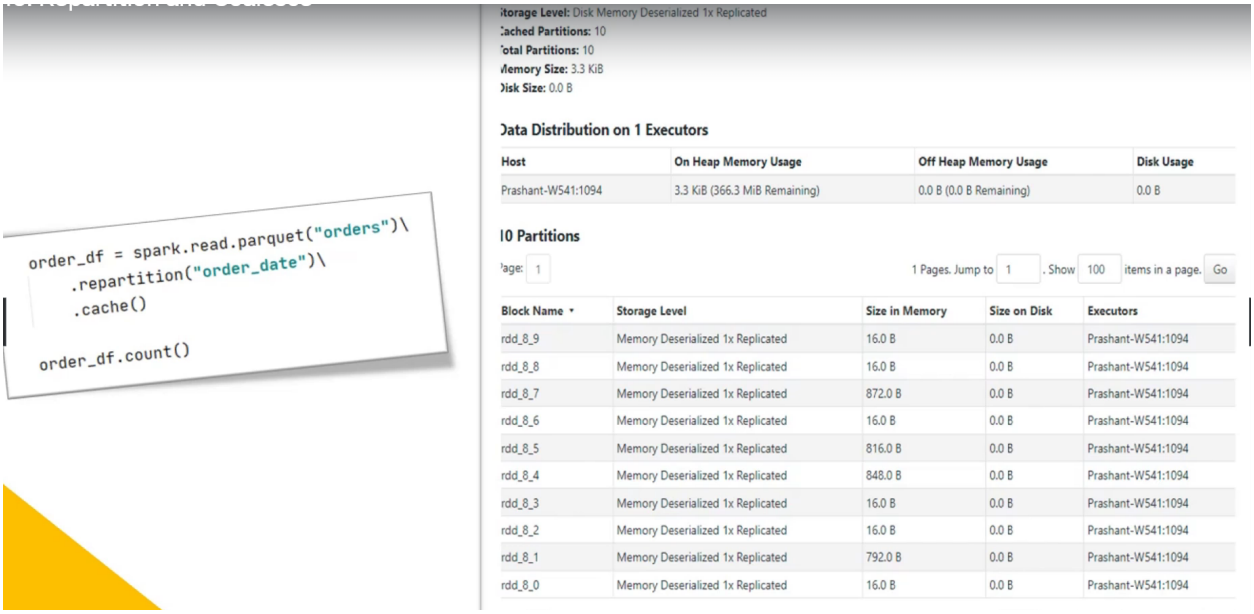
```
df = spark.range(1, 1000000).toDF("id") \
    .repartition(10) \
    .cache()

df.count()
```



From above screenshot , we are creating a data frame of 1 million rows , repartition it to 10 uniform partitions and cache it. Why cache? Because we want to see them in spark UI and learn how repartition behaves. What do we see on spark UI? We have 10 partitions and all are of same size(Size In Memory column) . So, repartition(10) creates 10 uniform partitions.

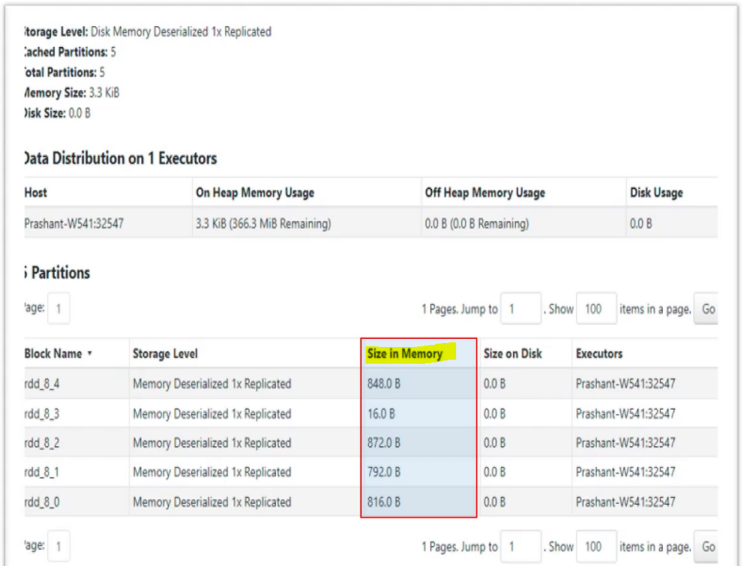
Now let's create another example , how repartition works for a column.

We are reading parquet data source and repartitioning it using order_date. And we got 10 partitions once again. But how? Repartition is a wide dependency transformation. So it will cause shuffle/sort and repartition our data using the order_date column. But no of partitions is controlled by spark.sql.shuffle.partitions configuration. We configured spark.sql.shuffle.partitions to 10 so we got 10 partitions.But we can override this behavior by supplying number of partitions.



The below is another example . This time we want only 5 output partitions and also want those partitions based on a column name. So spark will partition my data on the order_date and limit the no of output partitions to 5. But we can also see that partition size is not uniform as shown in below screenshot. So, repartition on a column name doesnot guarantee a uniform partitioning.

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

order_df.count()
```



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

i 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

repartition(numPartitions, *cols)

Summary

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

WHEN DO WE WANT TO REPARTITION OUR DATA FRAME?

Repartition will cause shuffle/sort , which is an expensive operation. We should always almost try to avoid unnecessary shuffle/sort. So, repartitioning our data frame must be done with caution. If we do not see any benefit of repartitioning , we must avoid it. And lets say few situations for repartitioning our data frame. We want to reuse data frame multiple times and filter our data on some specific column. In that case, repartitioning on filter column could be worth of it.

We have few partitions and our data is not well distributed across our cluster. In that case, repartitioning with large number of partitions can spread our data more appropriately across cluster and speed up the rest of our processing. Our partitions are large or we have some skewed partitions. In that scenario, we may want to uniformly repartition our data frame to a large number of uniform partitions. So repartitioning could be helpful in few scenarios. But we should make sure that we are getting benefits and possibly test it.

When should you repartition?

1. Repartition causes shuffle
2. Do not repartition without a reason
3. Common Repartition Scenarios
   - Dataframe reuse and repeated column filters
   - Dataframe partitions are not well distributed
   - Large Datarrame partitions or skewed partitions
- Do not repartition for reducing number of partitions

We should use repartitioning when we want to increase the number of partitions or repartition on specific columns. We should not use repartitioning to reduce number of partitions. Instead, we can use coalesce method for reducing number of partitions.

The below is the structure of coalesce method.

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 we have a 10-node cluster. And we have a data frame of 20 partitions. Those 20 partitions are spread on these 10 worker nodes.

Now we want to reduce number of partitions to 10.so we executed coalesce(10) on our data frame. So, spark will try to coalesce local partitions and reduce our partition count to 10. So final stage might look like this.

1. Coalesce doesn't cause a shuffle/sort
2. It combines local partitions only
3. Coalesce will not increase the number of partitions
4. Coalesce can cause skewed partitions

If we try to increase no of partitions using coalesce , it will do nothing. We must use repartition to increase number of partitions. we can also reduce our partition count using repartition() , but it will cost us a shuffle operation.

Coalesce can cause skewed partitions. So, try to avoid drastically decreasing number of partitions. It can cause skewed partitions, which may lead to an OOM exception.

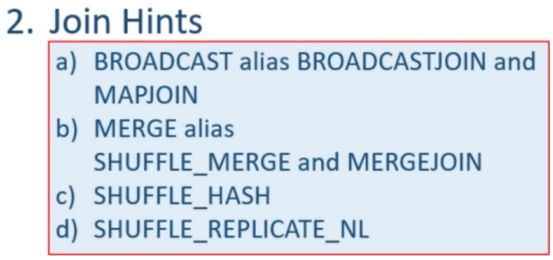
DATAFRAME HINTS

Spark will allows us to add 2 types of hints to our data frames and spark SQL. Partitioning hints and join hints.

We have 4 partition hints. Coalesce hint can be used to reduce number of partitions to the specified number of partitions. It will take partition number as a parameter.

Repartition hint can be used to repartition our data frame to the specified number of partitions. It takes a partition number, column names or both as parameters. Repartition_by_range is similar to repartition, but it uses data range for partitioning.

Rebalance hint was added after spark 3.0 . 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. This hint is helpful when we need to write the result of this query to a table to avoid too small/ too big files. This hint is ignored if AQE is not enabled.



These hints allow us to suggest 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 we have listed above. The last one is known as shuffle and replicate nested loop join. These joint hints also have some alias. So broadcast and broadcast join and mapjoin are same. They are just alias for the same hint.

If we are using spark SQL, we can use these hints using the below syntax.

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;
```

We have 2 methods to apply hints in dataframe.

Using Hints in Spark Dataframe

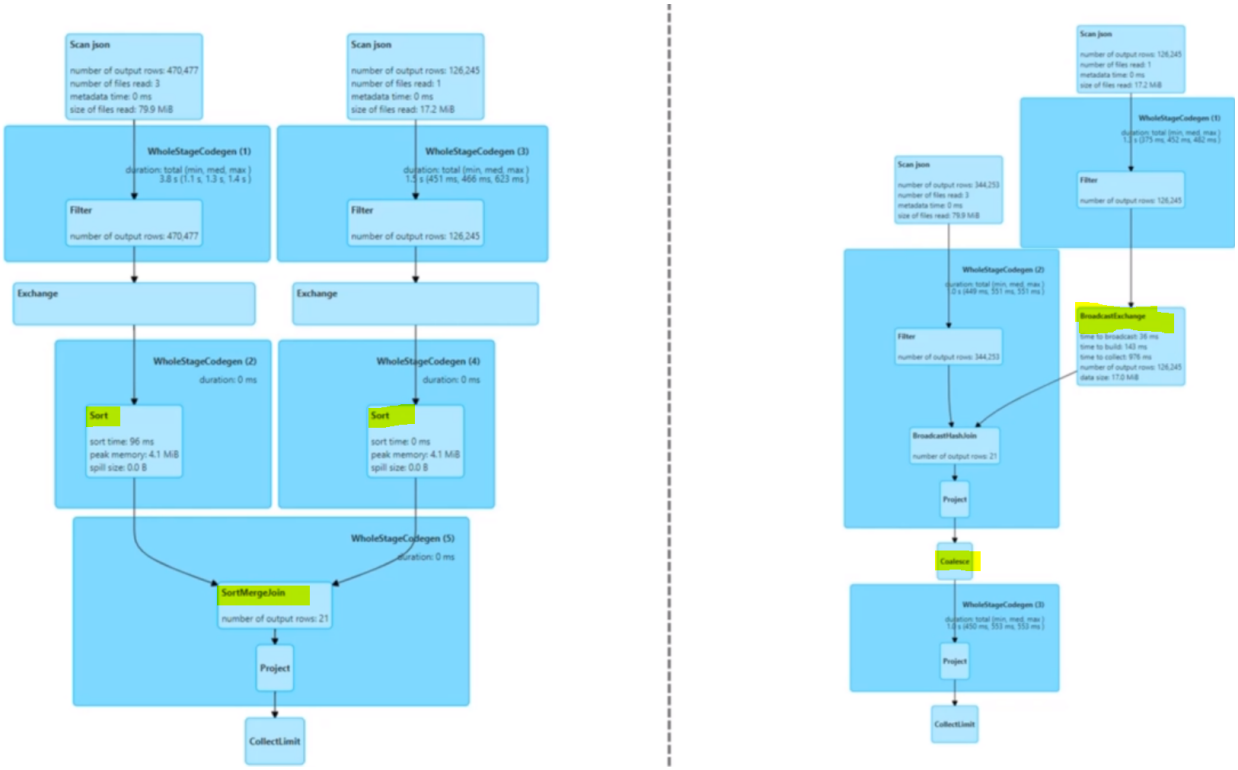
1. Use Spark SQL functions
2. use Dataframe.hint() method

```
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
```

https://github.com/ScholarNest/PySpark-Examples/tree/main/dataframe-hints

The above screenshot shows sample code where we used hints.

Df1 is larger data frame and df2 is smaller data frame. So wanted to apply broadcast hint to df2. So, I can avoid shuffle/sort. So that's why we applied broadcast function around the df2. broadcast() function is one approach to apply broadcast hint to a dataframe. Finally we are joining 2 data frames and applying a coalesce hint to the join result. Default value of shuffle partition is 200. So if we donot apply coalesce hint, join_df will have 200 partitions. But we applied coalesce(5) hint. So spark will coalesce it into 5 partitions only. We executed above code with out these 2 hints and also executed them with both the hints. The below are both execution plans. The left side plan is for without hints. And the right-side plan is for both the hints. We can see first plan(LHS) applies shuffle sort and then sort-merge join. Second plan applies a broadcast join and also applies a coalesce in the end.



SPARK BROADCAST VARIABLES

Spark broadcast variables are part of spark low-level API's. If we are using spark data frame API's , we are not likely to use broadcast variables. They are primarily used with spark low-level RDD API's .

Let's take an example. We want to create a spark UDF and use it in my spark project. Spark UDF is nothing but a function that takes one or more columns as input and returns one transformed column as output.

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 above function step 1 takes a string product_code and returns a string product name. And for translating product code to a product name , function references small lookup table. Here (step 2) is lookup data structure set. We want to create UDF and provided it with 2 inputs. First input is a product_code which goes as a function argument. The second input(prdCode) is a large dataset that should be available to function as a lookup table. In these kind of requirements, we want to create and use UDF. But we also want to make available some reference data to spark UDF. We cannot pass this reference data to UDF as a function argument. So how do we provide it to UDF? One easy way is to keep lookup data in file, load it at runtime and share it with UDF. But how do we share it with UDF? We have 2 ways to do it. 1 is closure and 2 is broadcast.

Th below screenshot shows a code. We read lookup.csv file from the storage and brought it to spark driver as a python dictionary. Now 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. Now bdData is my broadcast variable. Now we have 2 variables here. prdCode and bdData. prdCode is a standard python dictionary variable. However, bdData is a broadcast variable. So we are creating dataframe at runtime. We will use data frame (data_list) to test my UDF.

In the following 2 lines , we are registering my UDF function as spark sql udf and applying it to data frame. This is a standard method for spark UDF. Now let's see UDF definition at top of code in screenshot. We are accepting product code as input and returning the product name from bdData broadcast variable. So, whats happening here? My UDF function will be called for each record/row in my data frame. While calling UDF we are passing product_code column to UDF. UDF will look for product_code in the lookup(in data_list) and return the corresponding product name. we are using broadcast variable in (return bdData.value.get(code)) But we can also use prdCode directly. Both approaches are going to work. But if we are using prdCode, we are using lambda closure. And if we are using bdData , we are using broadcast variable. But what is difference between 2 approaches? If we are using a closure, spark will serialize closure to each task. If we have 1000 tasks running on a 30-node cluster, our closure is serialized 1000 times. But if we are using a broad cast variable, spark will

serialize it once per worker node and cache it there for future usage. So we have 1000 tasks and 30 worker nodes. In that case , broadcast variable is serialized only 30 times, once per worker node.

```
def my_func(code: str) -> str:
    # return prdCode.get(code)
    return bdData.value.get(code)


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

    prdCode = spark.read.csv("data/lookup.csv").rdd.collectAsMap()

    bdData = spark.sparkContext.broadcast(prdCode)

    data_list = [("98312", "2021-01-01", "1200", "01"),
                 ("01056", "2021-01-02", "2345", "01"),
                 ("98312", "2021-02-03", "1200", "02"),
                 ("01056", "2021-02-04", "2345", "02"),
                 ("02845", "2021-02-05", "9812", "02")]
    df = spark.createDataFrame(data_list) \
        .toDF("code", "order_date", "price", "qty")

    spark.udf.register("my_udf", my_func, StringType())
    df.withColumn("Product", expr("my_udf(code)")) \
        .show()
```

s://github.com/ScholarNest/PySpark-Examples/tree/main/broadcast-for-udf

Summary:

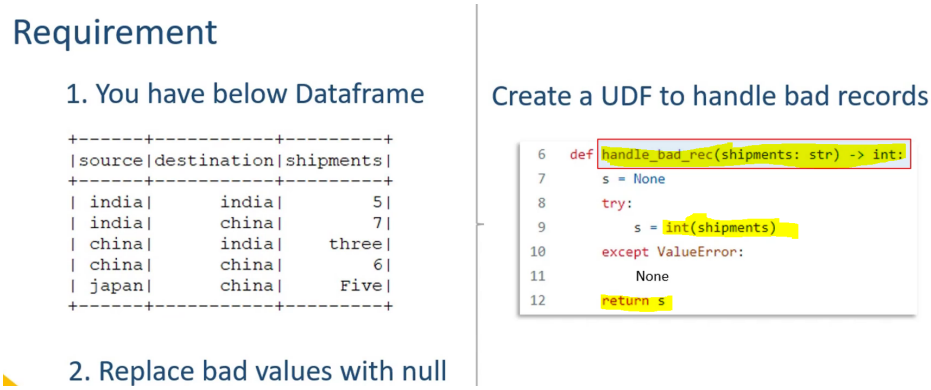
Broadcast variables are shared, immutable variables. Cached on every machine in the cluster instead of serialized with every single task. And this serialization is lazy which means broadcast variable is serialized to a worker node only if we have at least one task running on the worker node that needs to access broadcast variable. However we should also remember that broadcast variable must fit into the memory of an executor.

Spark dataframe API's uses this same technique to implement broadcast joins. So they will bring small table to the driver as we done in this above example. Then they will broadcast it to workers for being used in the join.

So, if we carefully design our application logic, we can meet our requirement using broadcast joins in most cases. However, if we plan to use UDF and make some datasets available to our UDF for reference, we can use the broadcast variables.

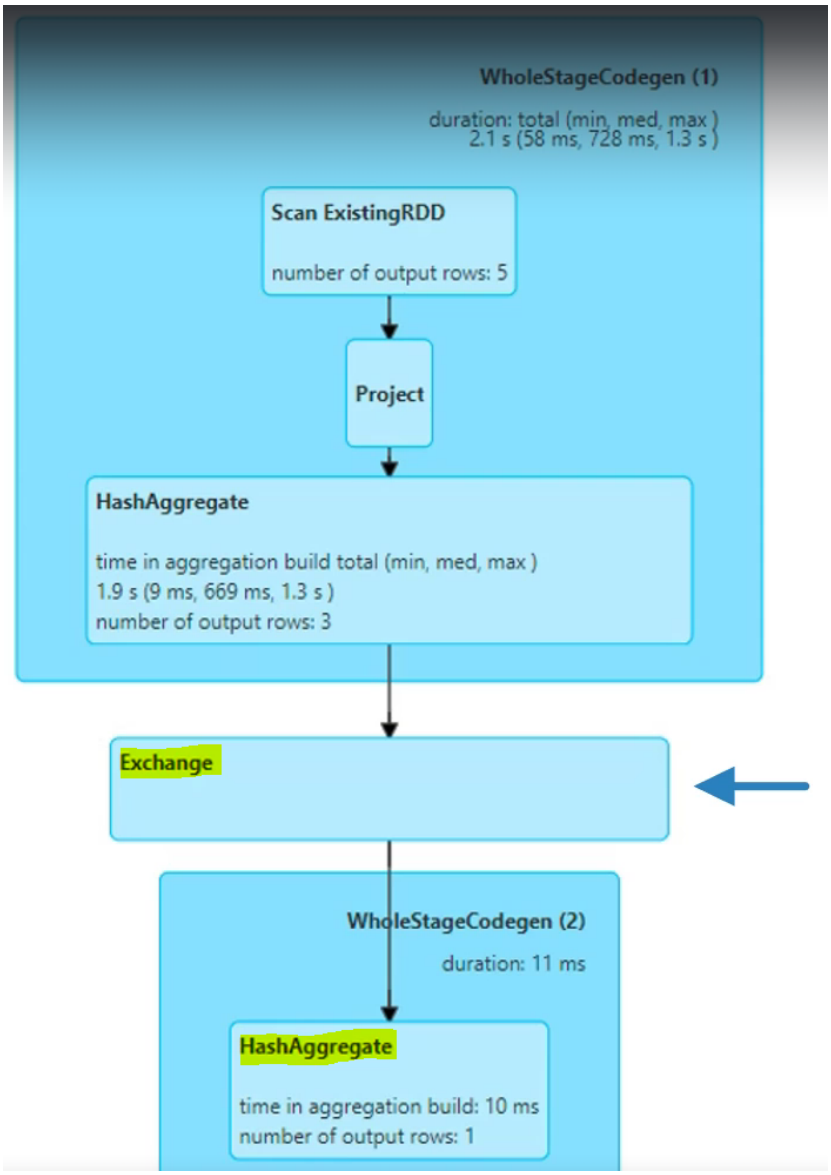
APACHE SPARK ACCUMULATORS

Spark Accumulators are part of spark low level API's. if we are using spark dataframe API's, we are not likely to use accumulators. Spark accumulators were primarily used with spark low-level RDD API's. Let's say I have an data frame that looks like below, This data frame represents an aggregated shipment record. In below data frame we have source column, destination column and shipments column. This shipment column represents no of shipments from given source to a destination. However we have a slight problem here. shipment column is expected to be an integer column. But we have some bad records in the table. We want to fix those bad records. How do we do this? We are asked to take null if the shipments count is not a valid integer.



So, we can do UDF for this and above is the code for the UDF. This UDF simply takes shipment column, converts it to an integer and returns it. If we cannot convert shipment column to an integer, we return null. So the below shows how to use UDF. So, we will register UDF and use it to create a new column with the correct values. The below is the expected output. Replaced bad values with nulls.

But we also have another requirement , we want to count number of bad records. But count() aggregation or a count() action on a data frame has a wide dependency. Spark will add one extra stage and shuffle operation. So, execution plan looks like below.



We can see exchange in above execution plan. This exchange represents a shuffle that is caused by count() operation and that's not a good thing. We don't want shuffle exchange in our plan. Can do it via accumulators.

Spark Accumulator is a global mutable variable that spark cluster can safely update on a per-row basis. We can use them to implement counters/sums.

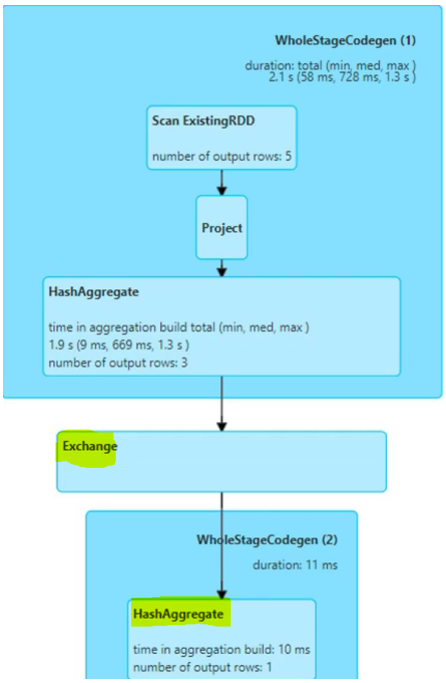
Let's see below code. We created an accumulator using spark context with an initial value of 0. So bad_rec is an accumulator variable. Then use this accumulator variable in my UDF. So, whenever I see bad record , I will increment accumulator by 1. This accumulator variable is maintained at the driver.

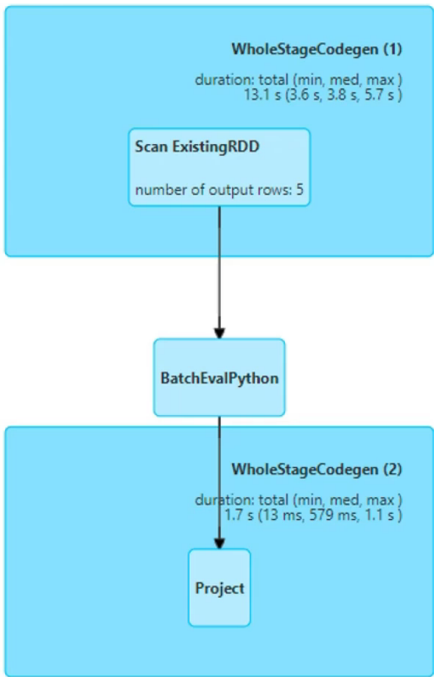
So we can simply print the final value. We do not have to collect it from anywhere. Because Accumulators always live at the driver.All the tasks will increment the value of this accumulator using internal communication with the driver. So, I donot have to include an extra stage and shuffle for calculating count of bad records. We are always incrementing it as I discover a bad record.

```python
def handle_bad_rec(shipments: str) -> int:
    s = None
    try:
        s = int(shipments)
    except ValueError:
        bad_rec.add(1)
    return s
```

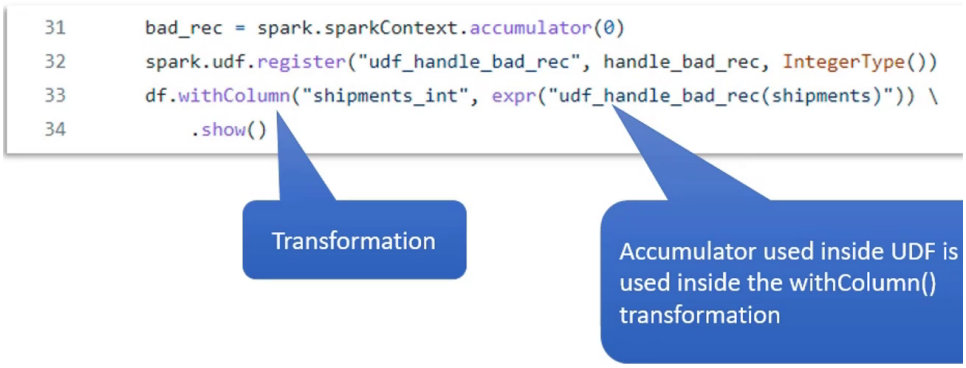
```python
if __name__ == "__main__":
    spark = SparkSession \
        .builder \
        .appName("Demo") \
        .master("local[3]") \
        .getOrCreate()

    data_list = [("india", "india", '5'),
                 ("india", "china", '7'),
                 ("china", "india", 'three'),
                 ("china", "china", '6'),
                 ("japan", "china", 'Five')]

    df = spark.createDataFrame(data_list) \
        .toDF("source", "destination", "shipments")

    bad_rec = spark.sparkContext.accumulator(0)
    spark.udf.register("udf_handle_bad_rec", handle_bad_rec, IntegerType())
    df.withColumn("shipments_int", expr("udf_handle_bad_rec(shipments)")) \
        .show()

    print("Bad Record Count:" + str(bad_rec.value))
```

Let's see below execution plan. The first is the old plan with count() and second uses an accumulator variable. So, we saved shuffle and hash aggregation while using accumulator variable. So that's one potential use of an accumulator. We can use accumulators for counting whatever we want to count while processing our data. They are similar to global counter variables in spark.





We used spark accumulator from inside withColumn() transformation. We used it from inside UDF but we are calling UDF inside the withColumn() transformation. So effectively accumulator is used inside the withColumn() transformation.



Some people used accumulator from inside an action. For example we also have for Each() method on pyspark dataframe. That's an action. forEach() action takes a lambda and applies lambda function on each row. We can also use an accumulator from within forEach() action. we can increment our 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. Because spark gives us a guarantee of accurate results when accumulator is incremented from inside an action. some spark tasks can fail due to variety of reasons. But 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 our accumulator multiple times , distorting our counter. But if we are incrementing our 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 we are incrementing our accumulator from inside a transformation , spark doesn't guarantees accurate results.

Spark in Scala also allow us to give our accumulator a name and show them in spark UI. However, pyspark accumulators are always unnamed, and they donot show up in the spark UI. Spark allow us to create long and float accumulators. However, we can also create some custom accumulators.

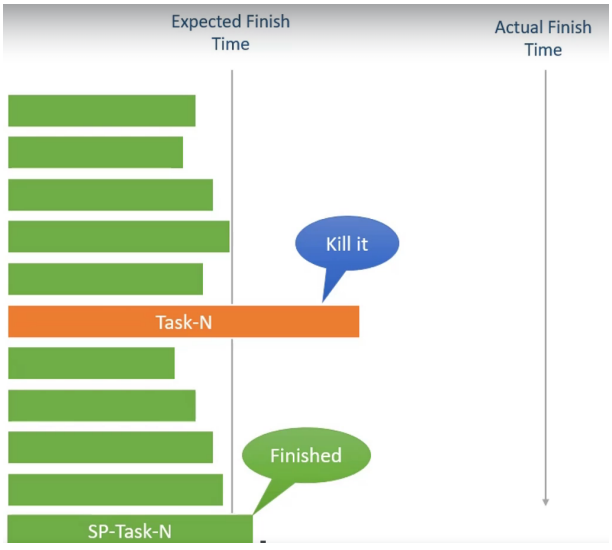
APACHE SPARK SPECULATIVE EXECUTION

Apache Spark runs our application as set of jobs. These jobs are broken down into stages and each stage is executed in parallel tasks. So, task is the smallest unit of work in apache spark. Let's assume our spark job is running 10 tasks. We looked at the spark UI and noticed that out of those 10 tasks, All other tasks take almost equal time to complete except one task. One task takes a much longer time to complete. However, spark job stage is not complete until all the tasks of that stage are complete. So, looking at diagram, one task is delaying our stage. If this task is also completed along with other tasks, our stage will complete faster, and we can reduce spark job execution time. So, what can we do ? One easy thing is to enable spark speculative execution. We can enable it by using spark.speculation = true . By default, spark.speculation is set to false. we can set it to true and enable it. Once we enable speculative execution, spark will automatically identify slow-running tasks and run a duplicate copy of

the same tasks on other nodes. The idea is to start the copy of the same task on another worker node and see if that runs faster. This new duplicate task is known as speculative task.

Now we have 2 copies of the task. So, spark will accept the task that finishes first and kill the other task. These speculative tasks can solve our 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 which means our task is 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 our task to run slow. If we restart that task on a different worker node, it might run faster. That's the main idea behind the spark speculative task execution.

Spark will automatically identify a slow running task , run the 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, we should also remember that enabling speculative execution will start taking some extra resources from our spark cluster. Running speculative tasks is an overhead and, in many cases, it might not give us any benefits. That's why speculative task is disabled by default.



A speculative task cannot help us in following below situations.

If our task is running slow due to data skew problem , it is slow because task is struggling with low memory. Spark doesn't know the root cause of slow running task and it simply starts the speculative task to see if that one finishes faster. So be careful before enabling speculative execution. If we have some extra resources , enabling speculation might be a good thing to do. Spark also offers some additional configurations to fine tune the speculation.

Here are they.

Tuning Speculation

1. spark.speculation.interval = 100ms
2. spark.speculation.multiplier = 1.5
3. spark.speculation.quantile = 0.75
4. spark.speculation.minTaskRuntime = 100ms
5. spark.speculation.task.duration.threshold = None

Default value of speculation interview is 100ms so spark will check for task slowness and see if a speculative task is needed and spark will perform that check in a 100ms loop.

Default value of speculation multiplier is 1.5 . So, 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 task is 5 secs, then task can be speculative if and only if it takes more than 7.5 secs.

The 3rd configuration represents a fraction of tasks that must complete before speculation is enabled for a particular stage. 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.

The 4th configuration 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.

The last one is the hard limit of task duration , after which the scheduler would try to speculative run the task.

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

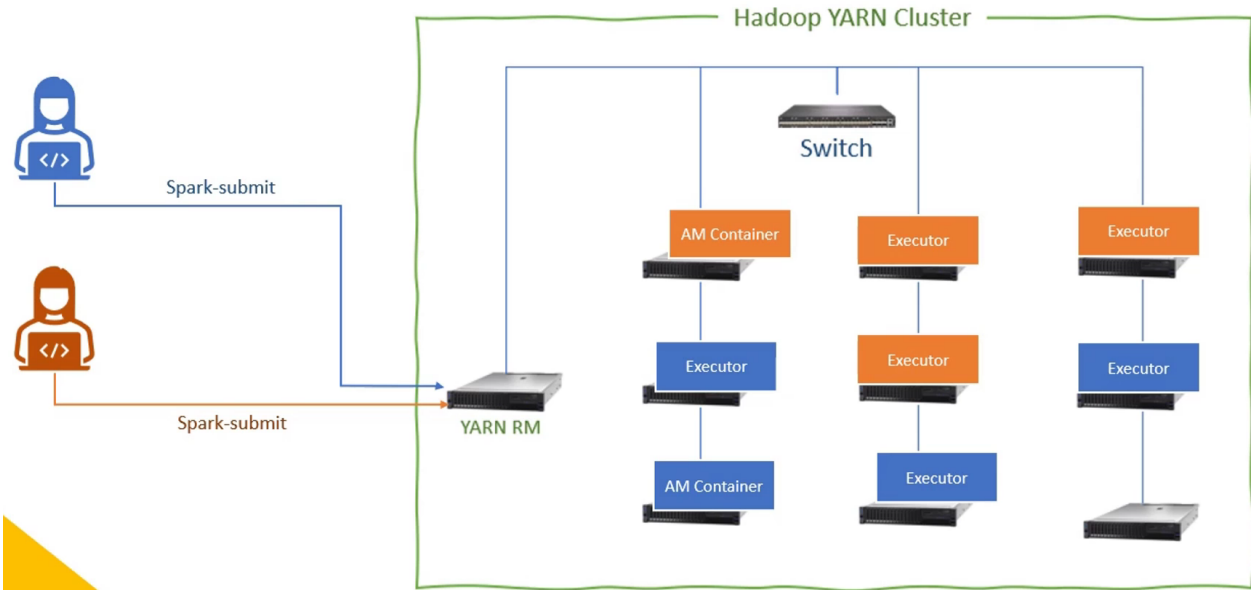
APACHE SPARK DYNAMIC RESOURCE ALLOCATION

When we talk about scheduling in the context of Apache spark, we mean 2 things. Scheduling Across Applications and Scheduling Within an Application.

SCENARIO OF SCHEDULING ACROSS APPLICATIONS:

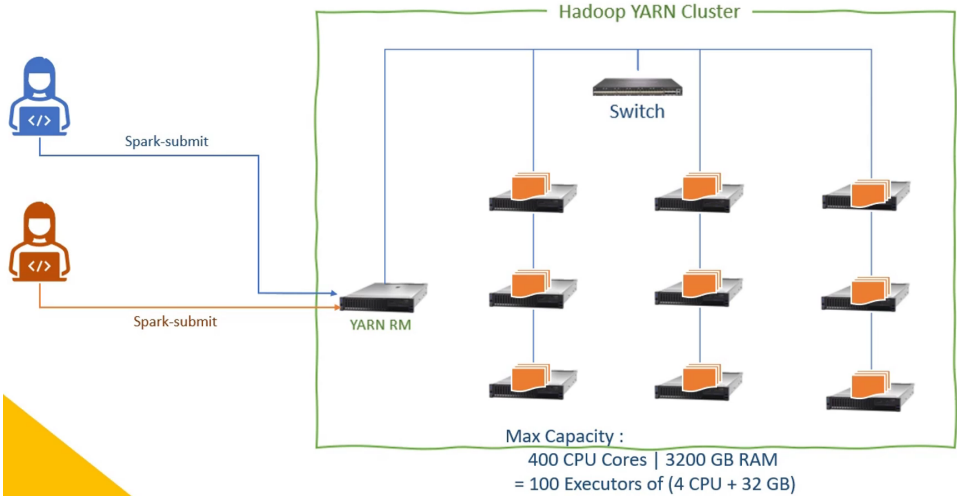
Spark is a distributed application and it runs on a cluster. So, we submit an application to a cluster. Cluster manager allocates some resources, and our application starts on the cluster. What if we

submit 2 applications to the same cluster or what if some other user also submits an application to the same cluster? 2nd application will also run on the same cluster. At any time, our cluster might be running more than one spark applications. So, in typical case spark cluster is shared by multiple applications.

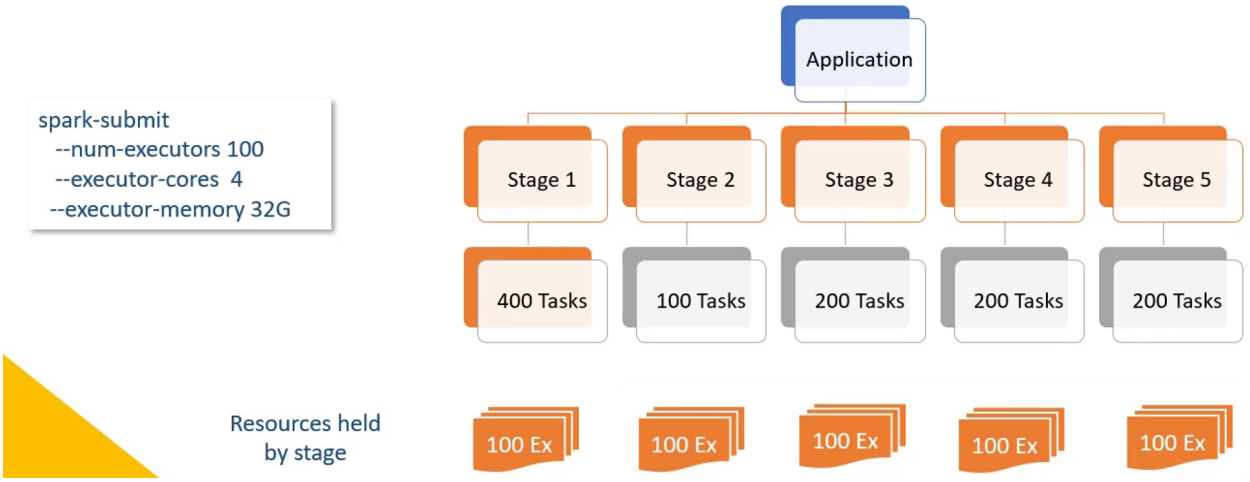


HOW CLUSTER MANAGER ALLOCATES RESOURCES TO THE SPARK APPLICATION AND WHEN DO THEY RELEASE THOSE RESOURCES?

Let's assume we have a cluster. And maximum capacity of the cluster is to create 100 containers. So, we cannot have more than 100 containers at a time. Now we started one application and application demands 100 containers. Resource manager will allocate 100 containers to the first application. What if we submit another application? 2nd application is small and it needs only 5 containers. But we donot have any containers left, so 2nd application must wait for some resources to become free. That's not efficient. Small application has to wait for a large application to finish. And this is an area where spark offers us some customization so we can better manage our resources.



Spark offers us 2 resource allocation strategies. Static Allocation and Dynamic Allocation. These 2 strategies are not for cluster Resource managers. These are for spark. They do not have anything to do with cluster resource manager. These 2 approaches define how our application requests resources and how it releases resources back to the cluster manager.



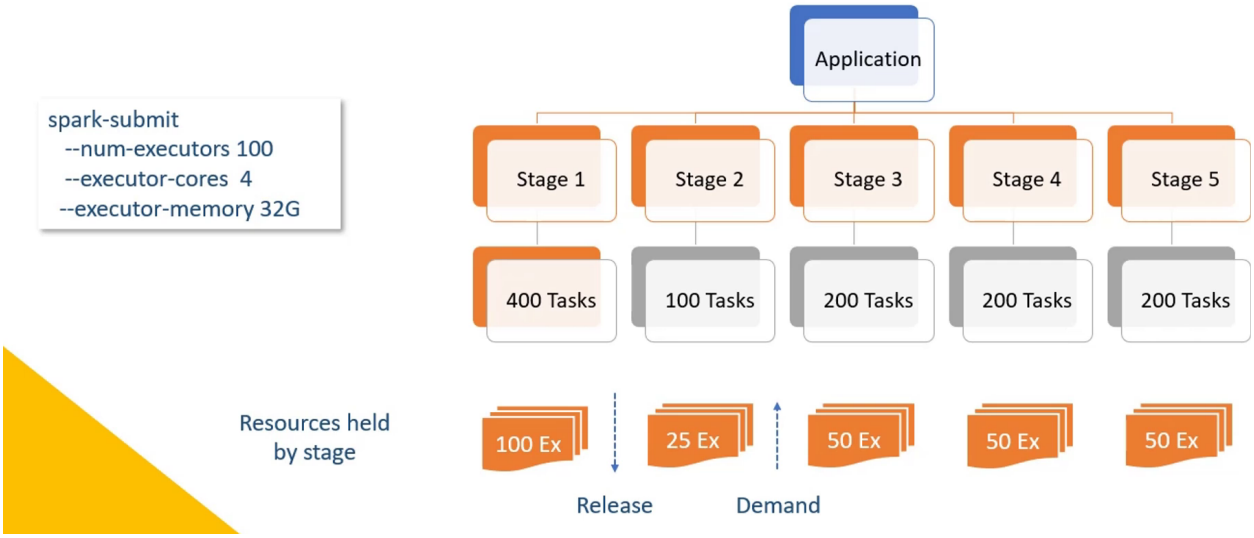
Static allocation is the default approach. In this approach as soon as our driver starts, it will request executor containers from the cluster resource manager. Cluster manager gives as many resources as demanded by the driver. But now our application will hold all those resources for the entire duration of running the application. Even if our application is not utilizing them, we will hold them with us. So, static allocation is a kind of first-come first reserve approach. In this approach, 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. This approach has a problem of resource utilization. Which means let's think about our application. We created an application that runs 5 stages. 1st stage requires 400 parallel tasks ,2nd stage requires 100 parallel tasks, and all other stages requires 200 parallel tasks. we demanded 100 executors with 4 CPU cores each Because we have a stage that need to run 400 tasks. So, we need 400 slots and hence we demanded 100 executors each with 4 slots. Our estimation is correct but our application runs in 5 stages. Only 1 stage is making good use of 100 executors. Other 4 stages are not using all the executors. But we are holding all those executors till the end. That's the problem. We can solve it via dynamic resource allocation. Spark offers dynamic resource allocation and we can enable it using below configurations. Dynamic configuration is disabled by default. But we can enable it to set this 2 configurations to be true.

```
spark.dynamicAllocation.enabled = true
spark.dynamicAllocation.shuffleTracking.enabled = true
```

Once we enable dynamic allocation, our spark application will automatically release the executors if they are not using it. Similarly they can again acquire executors when they are needed. 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 our application starts doing it dynamically.

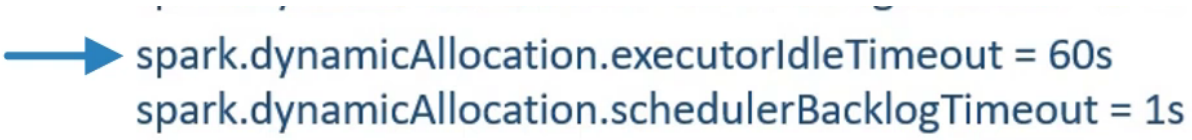
Let's see below example.



So, we need to run 400 parallel tasks in our 1st stage. So, we will demand 100 executors. The cluster manager will allocate our 100 executors and we will start running our 1st stage. Once we are done with our 1st stage , we need only 100 slots for the 2nd stage. So, we will release some executors back to the cluster manager because we donot need them for the moment. We finished our 2nd stage with few resources. But now we are into 3rd stage and we need more slots, So our application will again demand more slots and take them from the cluster manager.

Dynamic allocation is a spark configuration. It allows spark applications to release and demand resources as requirement changes dynamically.

We have 2 more important configurations.

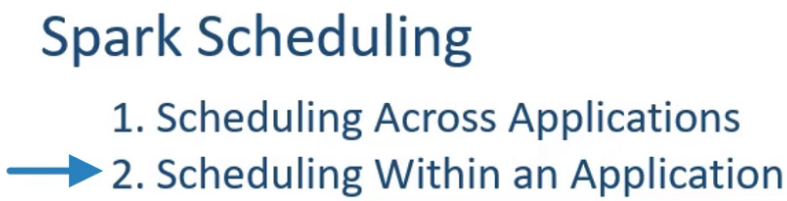


Default value of idle timeout is 60 seconds for above configuration . So, if an executor is idle and we don't have any task running on an executor for 60 seconds, our application will release the executor back to the cluster manager. We can change this configuration but 60 sec is a reasonable value for a standard application. So, this configuration controls the release time for an idle executor.

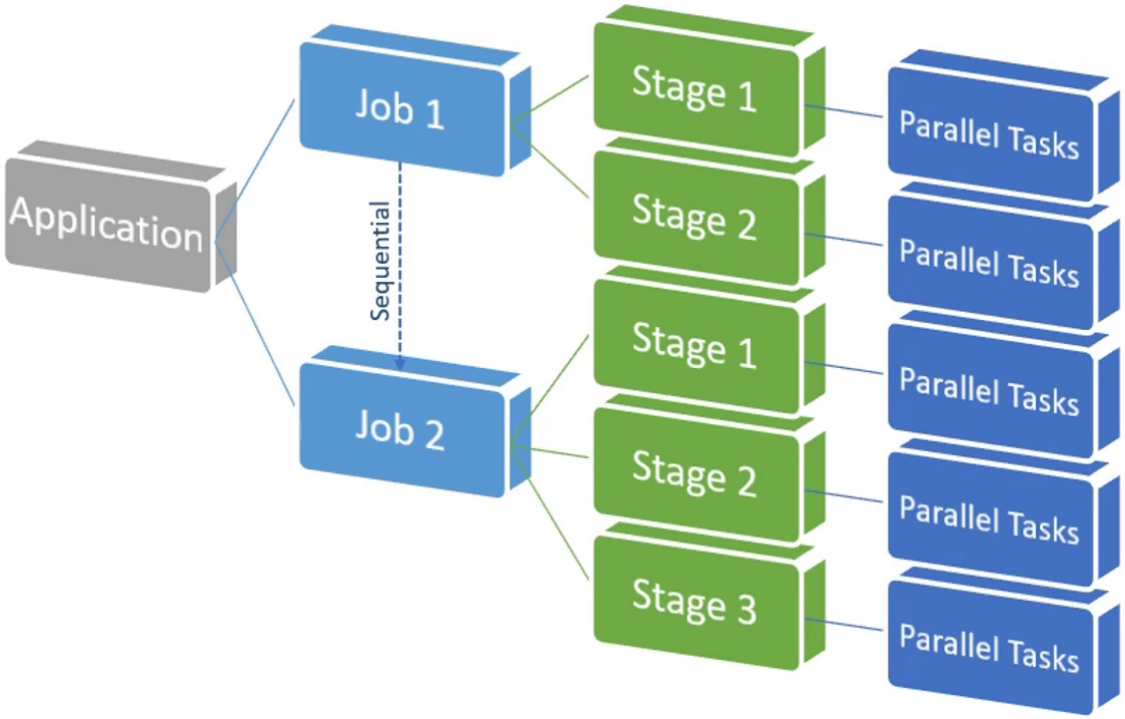
Default value for the backlog timeout is 1second. So, if we have some pending tasks for more than 1 second, and do not have any free executor where we can allocate our pending task. Our application will request more executors. This configuration controls the request time for more executors.

Cluster manager schedules spark application driver and resources for running the application are also provided by resource manager. Once our application driver starts , it can keep those resources or dynamically release and request them again. If we have a shared cluster, we should enable dynamic allocation so our spark applications can demand and release executors dynamically.

APACHE SPARK SCHEDULERS



Scheduling within spark application: We know spark application runs multiple jobs. Within single spark application, each of our actions triggers a spark job. In a typical case, these spark jobs run sequentially which means spark will start job1 and finish it. Then only job2 starts. That's the meaning of running spark jobs in a sequence.However, we also trigger them to run parallely.



Here the below is the example code. We are reading 2 datasets. The first one is df1 and the second one is df2. Then we joined df1 with df2 and took the count. We are also doing other same thing , reading data frame df3 and df4. Again, joining df3 with df4 , taking the count and printing it.

Here , we are doing 2 sets of independent operations. The first set of work will read df1, df2 , join them together and trigger a count() action. Similarly second set of work will read df3,df4 and join them together and trigger another count action. These 2 joins and counts are independent. The first

set of work has nothing to do with 2nd set of work. They do not have any dependency on each other. But my code will run them in a sequence.

```
from pyspark.sql import SparkSession

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

    df1 = spark.read.json("data/d1")
    df2 = spark.read.json("data/d2")
    print(df1.join(df2, "id", "inner").count())

    df3 = spark.read.json("data/d3")
    df4 = spark.read.json("data/d4")
    print(df3.join(df4, "id", "inner").count())
```

When we run the code, below jobs are triggered. We ran 6 jobs starting from job id 0 to job id 5. Now if see submitted timestamp, job 0 started at 16:26:43 and it took 2 seconds to complete. Job1 started 2 secs later when job 0 finished. Job 1 started at 16:26:45 which is exactly 2 secs of the run time of the job 0. The other jobs also started one after the other. So spark will run our 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 in our code block as shown above, we have 2 independent code blocks. The below one is the 2 independent code blocks.

```
df1 = spark.read.json("data/d1")
df2 = spark.read.json("data/d2")
print(df1.join(df2, "id", "inner").count())
```

```
df3 = spark.read.json("data/d3")
df4 = spark.read.json("data/d4")
print(df3.join(df4, "id", "inner").count())
```

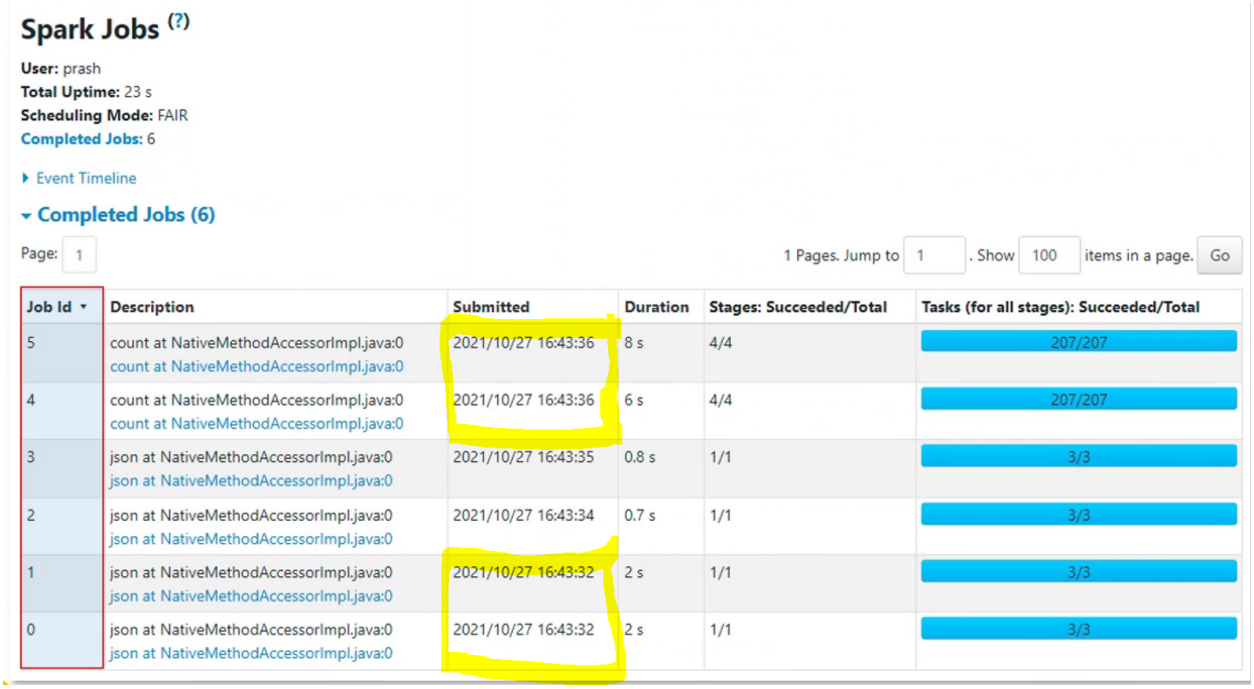
We prefer to run these 2 code blocks in parallel. Can we do it? Yes. All we have to do is to create 2 parallel threads and trigger each of these blocks from 2 different threads. We have to change the code and implement multithreading in my application. The below is the new code.

```
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)
```

We created do_job function as shown in LHS and this function takes 2 input parameters. These parameters are nothing but data file names. So I will read first data file into df1 , then the second data file into df2 , join these 2 data frames and count it. That's we had done in above code block in LHS. Now we come back to main method (RHS) and we will run the for loop(for i ------) . for loop runs twice. Every iteration will create one thread. The target for the thread is the same do_job function. But we are changing the file names for each iteration. So we are creating 2 threads here. Finally, we will start both threads and then wait for both the threads to complete. This code is designed to run do_job function in 2 different threads and both threads will trigger a count() action.

Now let's run the above code and below jobs details are shown.



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

We still have 6 jobs starting from job id 0 to job id 5. But now things are running in parallel. Job 0 and job 1 started in parallel at the same time. Job 4 and Job 5 also started in parallel at the same time. Our spark application runs as a set of spark jobs. Each Spark job represents an action. We can submit these actions sequentially or we can also submit these actions using parallel threads. But when we ran spark jobs in parallel, we have to think about resourcing i.e.; 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 we create a multi-threaded 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 our application is single threaded spark application, we do not worry about scheduling within the application. But if we create a multi-threaded spark application and submit jobs from multiple parallel threads , job scheduling becomes an important part. How do we handle it?

FIFO : 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 2nd job gets priority.

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 head of the queue are large, then later jobs may be delayed significantly. However, we can change the FIFO scheduler and configure spark to use the FAIR scheduler.

FAIR: Under fair sharing , spark assigns tasks between jobs in a round-robin fashion. What does that mean ? 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 we enable FAIR scheduler, we can see the list of fair scheduler pools in our spark UI as shown below. We are using FAIR scheduler default pool and with in the pool , we have a FIFO scheduler.

We can go further down to create and configure multiple pools within the FAIR scheduler but that may not be necessary. Fair scheduler with default pool works well enough.