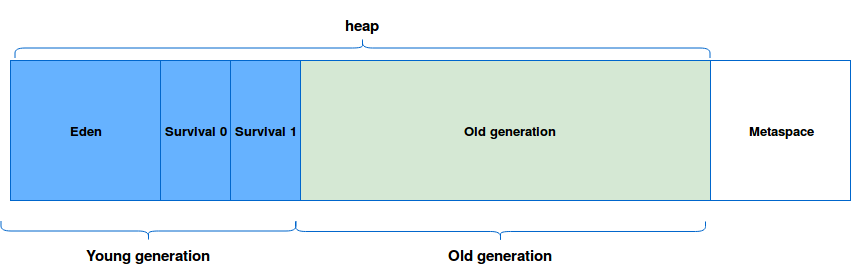
**ON-HEAP vs OFF-HEAP STORE**

The on-heap store refers to objects that will be present in the Java heap (and also subject to GC).

On the other hand, the off-heap store refers to (serialized) objects that are managed by EHCache, but stored outside the heap (and also not subject to GC). As the off-heap store continues to be managed in memory, it is slightly slower than the on-heap store, but still faster than the disk store.

**On-heap memory**

JVM-based applications use predefined amount of total system memory called on-heap that is divided in different spaces shown in image below:



Each newly created class instance is located in the eden space of young generation. If it's still in used in the moment of Garbage Collection, it's promoted to the next space. Finally, if it's a long-lived object, it's moved to the old generation where the GC are less frequent and more susceptible to produce longer stop-the-world events (old generation is usually much bigger than each of previous spaces). If you want to learn more about it, you can read the post about [Generations in JVM](https://www.waitingforcode.com/java-memory-model/generations-in-jvm/read).

After this short reminder we could characterize on-heap memory as a memory present in the JVM and managed automatically by the Garbage Collector.

**Off-heap memory**

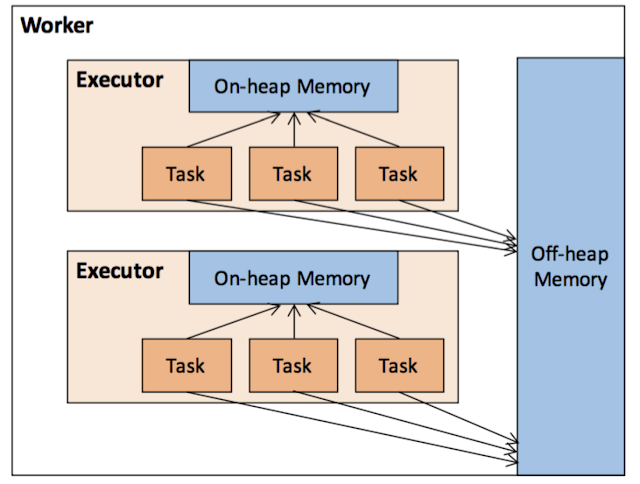
At first glance everything seems to be fine - objects are located and deallocated automatically, on-heap is freely configurable through appropriated options. But sometimes it's not enough, especially when we need to: cache a lot of data without increasing GC pauses, share cached data between JVMs or add a persistence layer in memory resistant to JVM crashes. In all mentioned cases off-heap memory is one of possible solutions.

As you can imagine, the off-heap memory stores the data outside the heap in OS memory part. Because there are no the JVM, the data must be stored in specific format that is an array of bytes. So using the off-heap memory in JVM languages programs introduces the overhead of serializing/deserializing these arrays to corresponding objects every time with additional cost of going outside the JVM and dealing with native memory. And because this space is out of JVM it can follow its own rules and bring other problems to programmers as *Big-Endian and Little-Endian one*.

However, the use off-heap can help to reduce GC pauses (especially in large heaps). It also allows different process to share the data stored in memory (e.g. C++ program and Scala one). In additional, off-heap memory helps the data to survive JVM crashes. With that it's possible to have a long living hot cache.

But off-heap memory is not the solution in all cases:

* Still short-lived objects (= never promoted to old generation) are better suited for on-heap storage simply because of the simplicity guaranteed by this automatic management.
* Moreover, the JIT can make several optimization for memory use (e.g. some objects allocation can be skipped thanks to [Escape Analysis](http://docs.oracle.com/javase/7/docs/technotes/guides/vm/performance-enhancements-7.html#escapeAnalysis)).
* In addition, off-heap storage involves serialization/deserialization overhead (we can save only arrays of bytes) that doesn't exist in on-heap objects storage.
* Off-heap storage means the manual management of the memory. Sometimes it can lead to memory leaks, seg faults or other uncommon problems in the life of "on-heap Java programmer".



**Use cases in Apache Spark**

Off-heap storage is not managed by the JVM's Garbage Collector mechanism. Hence, it must be handled explicitly by the application. Another difference with on-heap space consists of the storage format. In on-heap, the objects are serialized/deserialized automatically by the JVM but in off-heap, the application must handle this operation. In such a case the data must be converted to an array of bytes. If you want to know a little bit more about that topic, you can read the [On-heap vs off-heap storage](https://www.waitingforcode.com/off-heap/on-heap-off-heap-storage/read) post.

Off-heap memory is used in Apache Spark for the storage and for the execution data. The former use concerns caching. The persist method accepts a parameter being an instance of StorageLevel class. Its constructor takes a parameter \_useOffHeap defining whether the data will be stored off-heap or not. To test off-heap caching quickly we can use already defined *StorageLevel.OFF\_HEAP*:

"a RDD" should "not be cached in off-heap memory because of misconfiguration" in {

val conf = new SparkConf().setAppName("[OFFH] Cache").setMaster("local[4]")

val sparkContext = SparkContext.getOrCreate(conf)

val rddOfNumbers = sparkContext.parallelize(Seq(1, 2, 3, 4, 5))

val logAppender = InMemoryLogAppender.createLogAppender(Seq("offheap, 1 replicas"))

**rddOfNumbers.persist(StorageLevel.OFF\_HEAP)**

rddOfNumbers.count()

logAppender.getMessagesText() should have size 4

logAppender.getMessagesText() should contain allOf("Level for block rdd\_0\_0 is StorageLevel(disk, memory, offheap, 1 replicas)",

"Level for block rdd\_0\_1 is StorageLevel(disk, memory, offheap, 1 replicas)",

"Level for block rdd\_0\_2 is StorageLevel(disk, memory, offheap, 1 replicas)",

"Level for block rdd\_0\_3 is StorageLevel(disk, memory, offheap, 1 replicas)")

}

Internally the engine uses the def useOffHeap: Boolean = \_useOffHeap method to detect the type of storage memory. However, the above snippet won't cache the data in off-heap memory.

As you can see, the cache were stored directly on disk. It's because we didn't define the amount of off-heap memory available for our application. In order to make it work we need to explicitly enable off-heap storage with *spark.memory.offHeap.enabled* and also specify the amount of off-heap memory in *spark.memory.offHeap.size*. After doing that we can launch the following test:

"a RDD" should "be cached in off-heap memory when 2 required configuration entries are defined" in {

val conf = new SparkConf().setAppName("[OFFH] Cache").setMaster("local[4]")

**.set("spark.memory.offHeap.enabled", "true").set("spark.memory.offHeap.size", "3048576")**

val sparkContext = SparkContext.getOrCreate(conf)

val rddOfNumbers = sparkContext.parallelize(Seq(1, 2, 3, 4, 5))

val logAppender = InMemoryLogAppender.createLogAppender(Seq("offheap, 1 replicas", "Will not store rdd"))

**rddOfNumbers.persist(StorageLevel.OFF\_HEAP)**

**rddOfNumbers.count()**

logAppender.getMessagesText() should have size 4

logAppender.getMessagesText() should contain allOf("Level for block rdd\_0\_0 is StorageLevel(disk, memory, offheap, 1 replicas)",

"Level for block rdd\_0\_1 is StorageLevel(disk, memory, offheap, 1 replicas)",

"Level for block rdd\_0\_2 is StorageLevel(disk, memory, offheap, 1 replicas)",

"Level for block rdd\_0\_3 is StorageLevel(disk, memory, offheap, 1 replicas)")

}

When a RDD is cached in off-heap memory, the transformation from object into array of bytes is delegated to BlockManager and its *putIteratorAsBytes[T](blockId: BlockId, values: Iterator[T], classTag: ClassTag[T], memoryMode: MemoryMode)* method. The translation process is made by SerializedValuesHolder which resolves the allocator from memory mode in that way:

val allocator = memoryMode match {

case MemoryMode.ON\_HEAP => ByteBuffer.allocate \_

case MemoryMode.OFF\_HEAP => Platform.allocateDirectBuffer \_

}

Another use case is execution memory. A task may need some memory from the execution pool in order to store intermediate results. For example, the following snippet tries to use RowBasedKeyValueBatch to prepare data for aggregation:

"too small available off-heap memory" should "fail the task" in {

val memory1gb = "1000000000"

val TestSparkSession = SparkSession.builder().appName("[OFFH] Execution memory in Dataset").master("local[\*]")

**.config("spark.memory.offHeap.enabled", "true")**

**.config("spark.memory.offHeap.size", memory1gb)**

.getOrCreate()

val logAppender = InMemoryLogAppender.createLogAppender(Seq("VariableLengthRowBasedKeyValueBatch"))

import TestSparkSession.implicits.\_

val TestedDataSet = Seq(

("pl", "Scala", "Team1", 10), ("pl", "Java", "Team1", 1), ("pl", "C++", "Team2", 2),

("us", "Scala", "Team2",15), ("us", "Java", "Team2",3),

("fr", "Scala", "Team2",5), ("fr", "Java", "Team2",9)

).toDF("country", "language", "team", "projects\_number")

TestedDataSet.rollup($"country", $"language", $"team").sum("projects\_number").collect()

val executionReservedMemory = logAppender.getMessagesText()

.filter(message => message.contains("Task") && message.contains("acquired") &&

message.contains("for org.apache.spark.sql.catalyst.expressions.VariableLengthRowBasedKeyValueBatch"))

executionReservedMemory should not be empty

val executionReleasedMemory = logAppender.getMessagesText()

.filter(message => message.contains("Task") && message.contains("release") &&

message.contains("from org.apache.spark.sql.catalyst.expressions.VariableLengthRowBasedKeyValueBatch"))

executionReleasedMemory should not be empty

}

However, defining the use of off-heap memory explicitly doesn't mean that Apache Spark will use only it. The framework also reserves the on-heap memory. In the previous examples, we can observe the use of on-heap memory for the closures defining the processing logic. In such a case, and at least for local mode (cluster mode will be detailed in the last part), the amount of on-heap memory is computed directly from runtime memory, as:

private def getMaxMemory(conf: SparkConf): Long = {

val systemMemory = conf.getLong("spark.testing.memory", Runtime.getRuntime.maxMemory)

val reservedMemory = conf.getLong("spark.testing.reservedMemory",

if (conf.contains("spark.testing")) 0 else RESERVED\_SYSTEM\_MEMORY\_BYTES)

val minSystemMemory = (reservedMemory \* 1.5).ceil.toLong

val usableMemory = systemMemory - reservedMemory

val memoryFraction = conf.getDouble("spark.memory.fraction", 0.6)

(usableMemory \* memoryFraction).toLong

}

val maxMemory = getMaxMemory(conf)

new UnifiedMemoryManager(

conf,

maxHeapMemory = maxMemory,

onHeapStorageRegionSize =

(maxMemory \* conf.getDouble("spark.memory.storageFraction", 0.5)).toLong,

numCores = numCores)

**MAIN:**

The reasons to use off-heap memory rather than on-heap are the same as in all JVM-based applications. It helps to reduce GC overhead, to share some data among 2 different processes, to have always ready-to-use cache data (even after tasks restart). However, it doesn't come without costs. In the flip side, the off-heap increases CPU usage because of the extra translation from bytes of arrays into expected JVM object.

The off-heap has also a trap. Even though we manage to store JVM objects off-heap, when they're read back to be used in the program, they can be allocated on-heap. Thus, there will be the need to garbage collect them. Therefore, in the Apache Spark context, in my opinion, it makes sense to use off-heap for SQL or Structured Streaming because they don't need to serialize back the data from the bytes array. The use in RDD-based programs can be useful though but should be studied with a little bit more care. Nonetheless, please notice that the Project Tungsten's format was designed to be efficient on on-heap memory too. Hence to decide whether go to on-heap or off-heap, we should always make the benchmark and use the most optimal solution only when the difference is big between them. Otherwise, it's always good to keep things simple and make them more complicated only when some important performance problems appear.

**Off-heap memory and Project Tungsten**

First and foremost, for me the most of confusion about off-heap and on-heap memory was introduced with Project Tungsten revolutionary storage format. Dataset stores the data not as Java or Kryo-serialized objects but as the arrays of bytes. Since this storage is intuitively related to the off-heap memory, we could suppose that it natively uses off-heap. But it's not true. Modules based on Project Tungsten, therefore Apache Spark SQL and Apache Spark Structured Streaming, will use off-heap memory only and only when it's explicitly enabled and when it's supported by the executor's JVM. The array-based storage format can help to reduce GC overhead though and it's even on the on-heap because there is rarely a need to serialize it back from compact array binary format.

**How to do performance tuning in spark?**

**Serialization**  
Any distributed application's performance is heavily influenced by serialization. Spark utilizes the Java serializer by default. Most Spark tasks work in a pipeline, with one Spark job writing data to a file, followed by another Spark job reading the data, processing it, and writing it to another file for another Spark job to read. When you have a use case like this, you should write an intermediate file in a serialized and optimized format like Avro, Kryo, Parquet, and so on, because any transformations on these formats perform better than text, CSV, and JSON.  
  
**Using DataFrame/Dataset over RDD**  
RDD, DataFrame, and DataSet are the three types of APIs available in Spark. RDD is a low-level operating system with few optimization strategies. In most circumstances, DataFrame is the best option since it employs the catalyst optimizer, which generates a query plan that improves performance. In addition, DataFrame has a low manpower garbage collection overhead. DataSets are very type-safe, and their serialization includes the encoder. It also makes use of Tungsten as a binary serializer.  
Because Spark doesn't know how to use optimization techniques, and RDD serializes and de-serializes data when it distributes over a cluster, using RDD directly causes performance concerns (repartition & shuffling). For Spark applications or other distributed systems, serialization and de-serialization are relatively expensive activities; we spend most of our time serializing data rather than executing actions, hence we strive to avoid utilizing RDD.  
  
**Caching and Persisting data**  
Persisting/caching in Spark is one of the most effective ways to boost the performance of Spark workloads. Spark provides an optimization technique to store the intermediate computation of a Spark DataFrame using the cache() and persist() methods so that they can be reused in subsequent actions. When you persist a dataset, each node saves its partitioned data in memory and reuses it in subsequent operations on the dataset. Spark's persisted data on nodes is fault-tolerant, which means that if a Dataset's partition is lost, it will be immediately recomputed using the original operations that formed it. When caching, use in-memory columnar format. You may further optimize Spark speed by tweaking the batchSize property. To store the cached data, Spark provides multiple storage levels; choose the one that best matches your cluster.  
  
**Reducing expensive shuffling operations**  
Spark utilizes a process called shuffling to disperse data among different executors and even machines. When we do specific transformation operations on RDD and DataFrame, such as groupByKey(), reducebyKey(), and join(), Spark shuffling occurs. Spark Shuffle is a costly procedure because it entails the following:

* Disk and Network I/O
* Data serialization and deserialization

A user can get the mistake out of memory if there is a lot of shuffling. To avoid this error, the amount of parallelism should be increased. We can't totally prevent shuffle operations, but we can try to decrease the amount of them and remove any that aren't being used. To customize the partitions of the shuffle, Spark provides the spark.sql.shuffle.partitions configurations. You can increase Spark performance by tuning this attribute.

**What is Spark DataFrame?**

DataFrames are distributed collections of data arranged into rows and columns in Spark. Each column in a DataFrame has a name and a type assigned to it. DataFrames are structured and compact, similar to standard database tables.

Spark DataFrames can be derived from a variety of sources, including Hive tables, log tables, external databases, and existing RDDs. Massive volumes of data may be processed with DataFrames. A Schema is a blueprint that is used by every DataFrame. It can contain both general data types like string types and integer types, as well as spark-specific data types such as struct types.  
  
DataFrames addressed the performance and scalability issues that arise when utilizing RDDs.  
  
RDDs fail to function properly when there is insufficient storage space in memory or on a disc. Furthermore, Spark RDDs lack the idea of schema, which is the structure of a database that defines its objects. RDDs hold both organized and unstructured data, which is inefficient.  
  
RDDs cannot alter the system to make it run more efficiently. RDDs do not allow us to debug issues while they are running. They keep the data in the form of a collection of Java objects.  
  
RDDs employ serialization (the act of turning an object into a stream of bytes to allow for faster processing) and garbage collection (an automatic memory management approach that discovers unneeded items and frees them from memory). Because they are so long, they put a strain on the system's memory.  
  
Let's take a look at what makes Spark DataFrames so distinctive and popular:

* Flexibility: DataFrames, like RDDs, can support a wide range of data formats which includes .CSV, Casandra, and many more.
* Scalability: DataFrames may be coupled with a variety of different Big Data tools and can process data ranging from megabytes to petabytes at once.
* Input Optimization Engine: To process data efficiently, DataFrames make use of input optimization engines, such as Catalyst Optimizer. The same engine can be used for any Python, Java, Scala, and R DataFrame APIs.
* Handling Structured Data: DataFrames provide a graphical representation of data. When data is stored in this manner, it has some meaning.
* Custom Memory Management: RDDs keep data in memory, whereas DataFrames store data off-heap (outside the main Java Heap region, but still inside RAM), reducing garbage collection overload.

**Transformations** Spark Transformation is a function that produces new RDD from the existing RDDs. It takes RDD as input and has one or more RDD as output. Each time it creates a new RDD when we apply any transformation. Using transformation, built an RDD lineage with the entire parent RDDs of the final RDD(s). RDD lineage, also known as RDD operator graph or RDD dependency graph. It is a logical execution plan, i.e., it is a Directed Acyclic Graph (DAG) of the entire parent RDDs of RDD. Transformations are lazy; they get executed when we call an action. They are not executed immediately.

**Action** transformations create RDDs from each other, but action is performed when we want to work with the actual dataset. When the action is triggered after the result, a new RDD is not formed like transformation. Thus, Actions are Spark RDD operations that give non-RDD values. The values of action are stored to drivers or the external storage system.

**Creation of Test DataFrame**

Here, we have created to DataFrames and performed inner join, and An action show(false) is performed. To get the final DataFrame after joining.

val employee = Seq((1,"ramu",3,"2018",10001,"M",25000), (2,"raju",1,"2010",20001,"M",40000), (3,"mahesh",1,"2010",10001,"M",35000), (4,"suresh",2,"2005",10001,"M",45000), (5,"likitha",2,"2010",40001,"F",35000), (6,"lavanya",2,"2010",50001,"F",40000), (8,"madhu",1,"2011",50001,"",40000))

val emp\_schema =

Seq("emp\_id","name","reporting\_head\_id","year\_joined","dept\_id","gender","salary")

val employeeDF = employee.toDF(emp\_schema:\_\*)

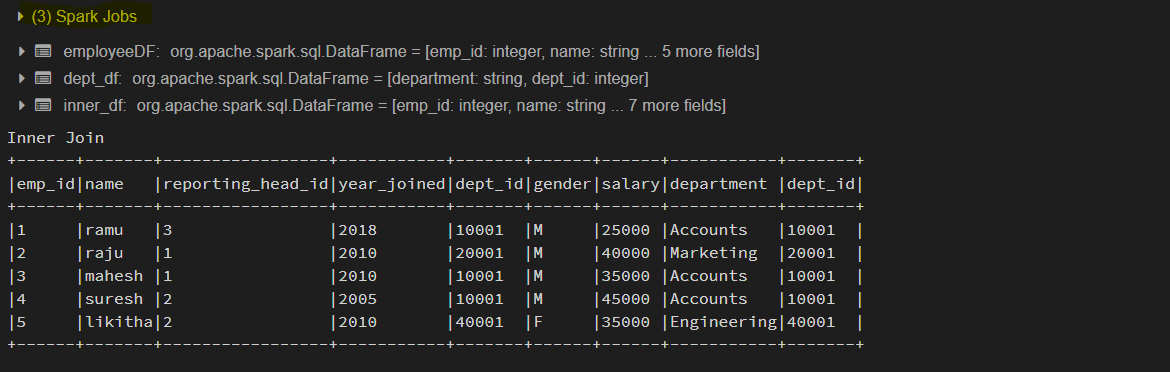
val dept = Seq(("Accounts",10001), ("Marketing",20001), ("Finance",30001), ("Engineering",40001) )

val dept\_schema = Seq("department","dept\_id")

val dept\_df = dept.toDF(dept\_schema:\_\*)

val inner\_df = employeeDF.join(dept\_df,employeeDF("dept\_id") === dept\_df("dept\_id"),"inner")

inner\_df.show(false)



Here we are going to study Spark UI in databricks. Spark UI is separated into tabs as below.

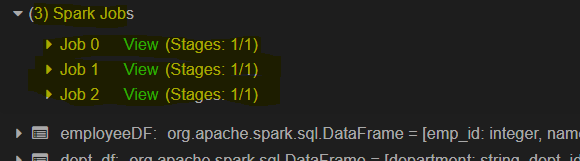
1. Spark Jobs
2. Stages
3. Tasks
4. Storage
5. Environment
6. Executors
7. SQL

1. Spark Jobs Tab

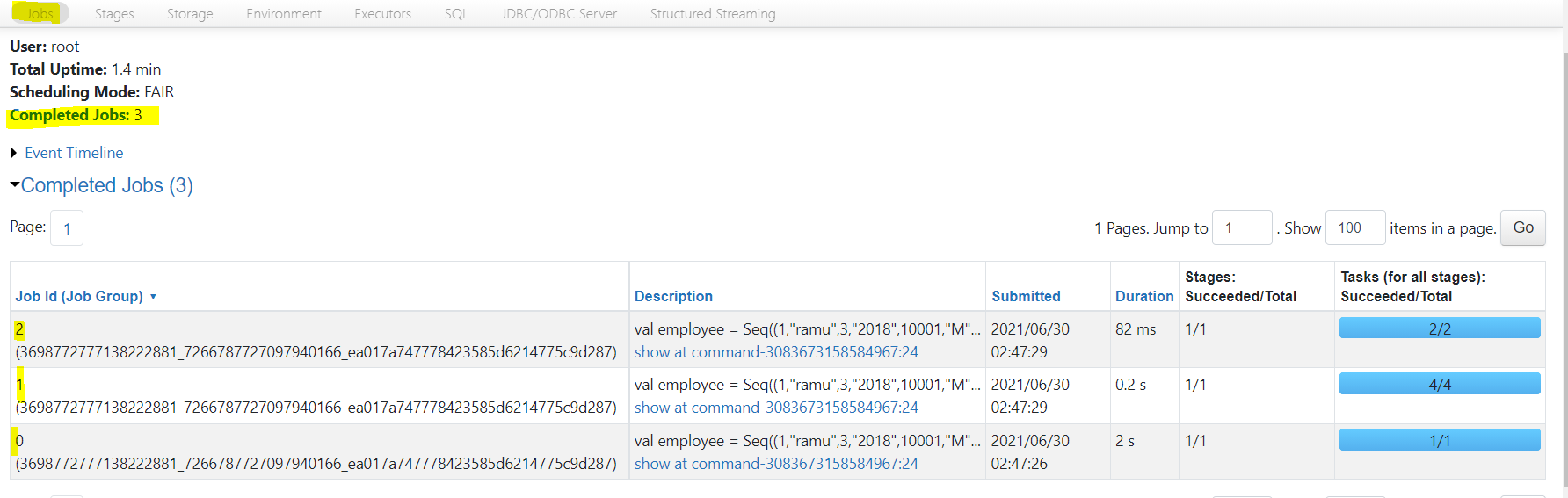
The details that I want you to be aware of under the jobs section are Scheduling mode, the number of Spark Jobs, the number of stages it has, and the Description of your spark job. Scheduling Mode -> In databricks, by default, the community version is provided with only one NODE.

**The Number of Spark Jobs ->** Always keep in mind, the number of Spark jobs is equal to the number of actions in the application, and each Spark job should have at least one Stage. In our above application, we have performed 3 Spark jobs (0,1,2)

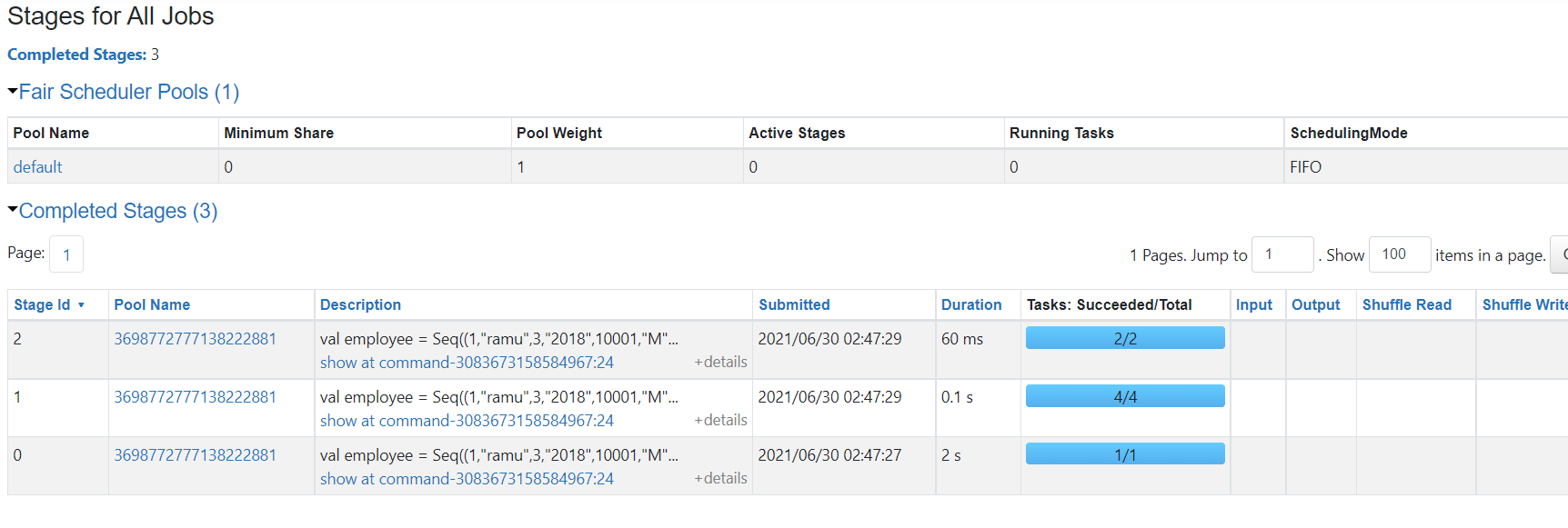
**Number of Stages ->**Each Wide Transformation results in a separate Number of Stages. In our case, Spark job0 and Spark job1, Spark job1 have single individual stages.



**Description ->** Description links the complete details of the associated SparkJob like Spark Job Status, DAG Visualization, Completed Stages. And below image shows how a job is to be in spark UI.



2. Stages Tab

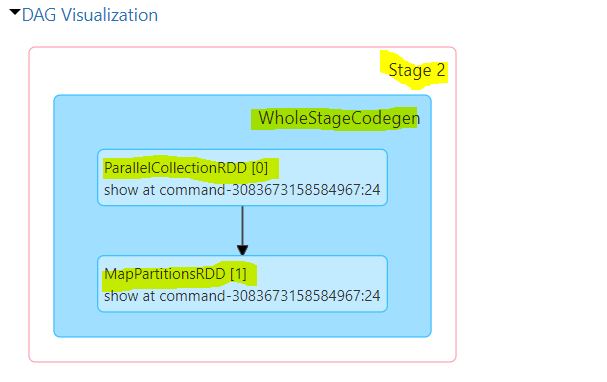


The above image gives the UI representation of the Spark Stage Tab. We can navigate into Stage Tab in two ways. One by Select the Description of the respective Spark job (Shows stages only for the Spark job opted).On the top of the Spark Job tab, select the Stages option (Which shows all stages in the application). In our application, we have a total of 3 Stages.

The Stage tab displays a summary page that shows the current state of all stages of all Spark jobs in the spark application.

The number of tasks you could see in each Stage is the number of partitions that spark is going to work on, and each task inside a stage is the same work that will be done by Spark but on a different partition of data.

Here In our case, we have MapPartitionsRDD, parallelcollection Rdd in wholestagecodegen Stage in all stages 0,1,2.As below



MapPartitionsRDD -> MapPartitionsRDD will be created when you use map Partition transformation

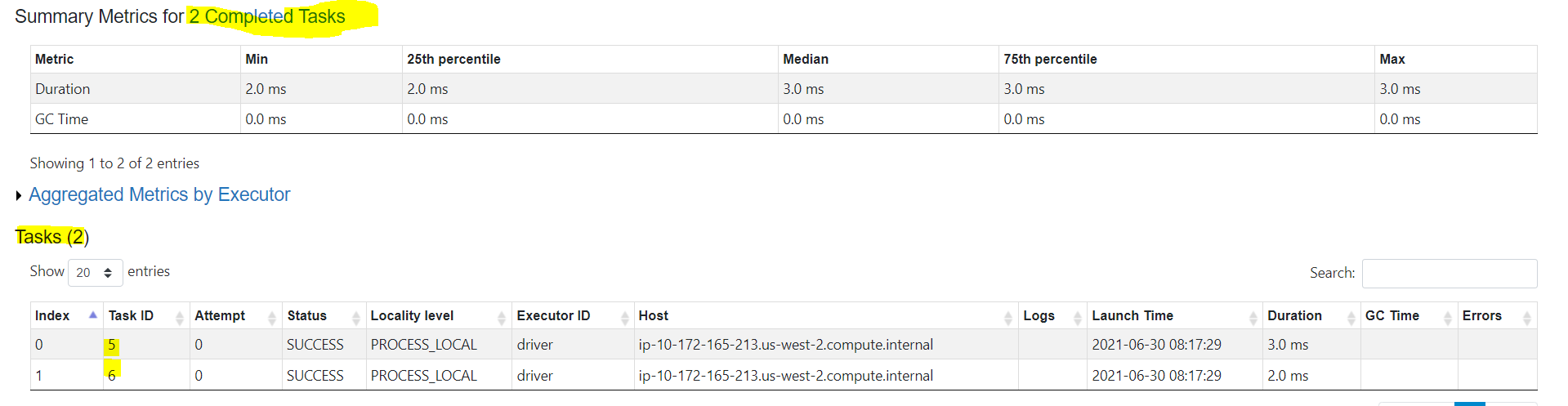
parallelcollectionRDD -> parallelcollectionRDD will be created when you use Parallelize method to make partitioned data.

Wholestagecodegen -> A physical query optimizer in Spark SQL that fuses multiple physical operators.

3. Tasks

Tasks are located at the bottom space in the respective Stage. Key things to look task page are

1. Input Size – Input for the Stage and 2. Shuffle Write-Output is the Stage written. The below image shows the tasks that are executed in stage 2 of our spark application execution.



4. Storage

The Storage tab displays the persisted RDDs and DataFrames, if any, in the application. The summary page shows the storage levels, sizes, and partitions of all RDDs, and the details page shows the dimensions and uses executors for all partitions in an RDD or DataFrame.

5. Environment

This environment page has five parts. It is an excellent place to check whether your properties have been set correctly.

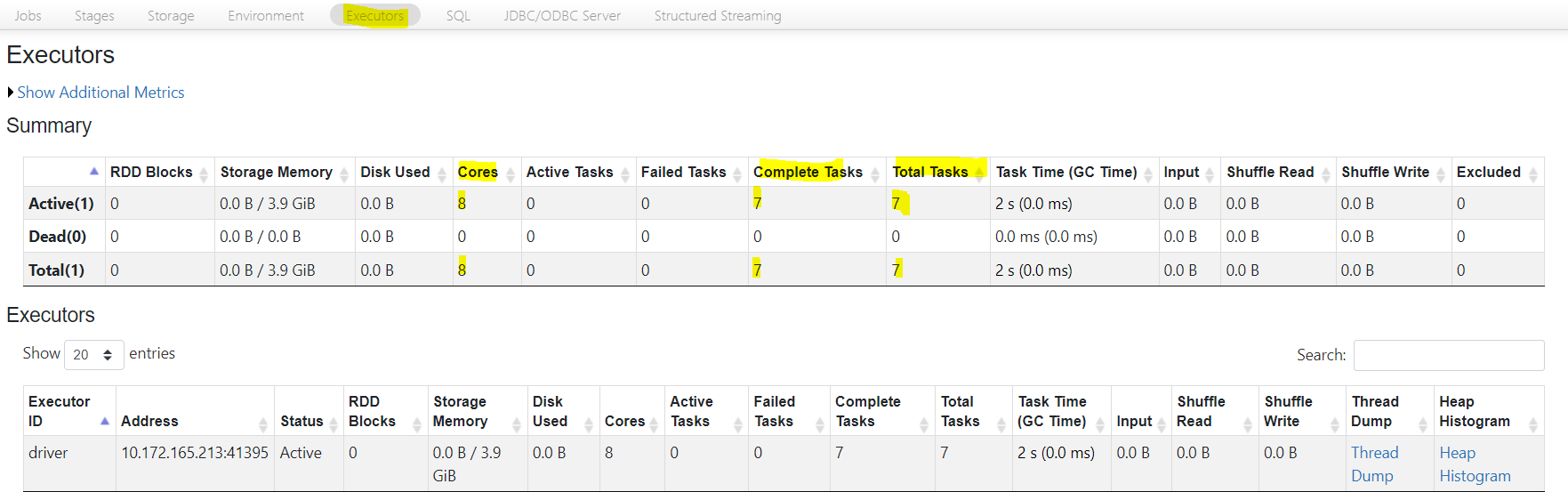
* Runtime Information: contains the runtime properties like versions of Java and Scala.
* Spark Properties: lists the application properties like ‘spark.app.name’ and ‘spark.driver.memory’.
* Hadoop Properties: displays properties relative to Hadoop and YARN. Note: Properties like ‘spark.hadoop’ are shown not in this part but in ‘Spark Properties.’
* System Properties: shows more details about the JVM.
* Classpath Entries: lists the classes loaded from different sources, which is very useful to resolve class conflicts.



6. Executors

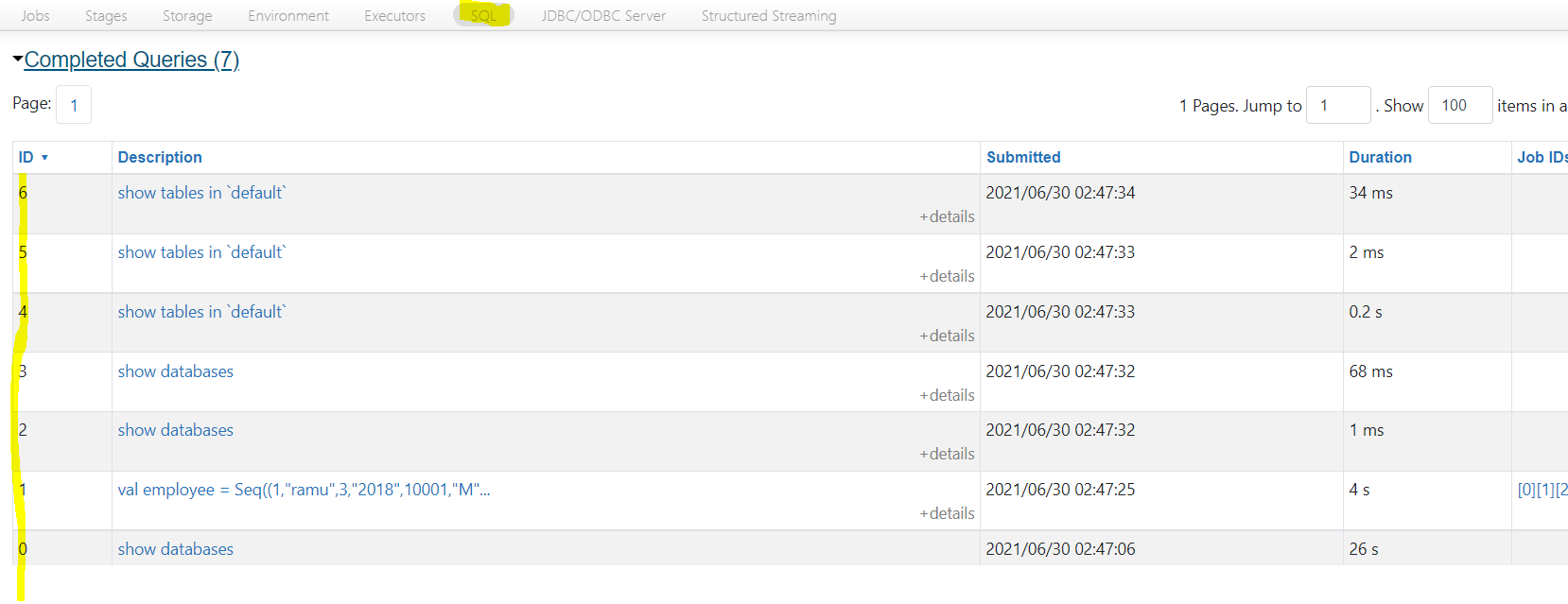
The Executors tab displays summary information about the executors created for the application, including memory and disk usage and task and shuffle information. The Storage Memory column shows the amount of memory used and reserved for caching data.

The Executors tab provides resource information like the amount of memory, disk, and cores of each executor and performance information. In our case, In Executors Number of cores = 8 and Number of tasks = 7.



7. SQL Tab

If the application executes Spark SQL queries, then the SQL tab displays information, such as the duration, Spark jobs, and physical and logical plans for the queries. In our application, we created a dataframe from raw data, and we joined those dataframes and called Show(). Here our spark application executed seven SQL queries.



**Explain Spark Catalyst Optimizer.**

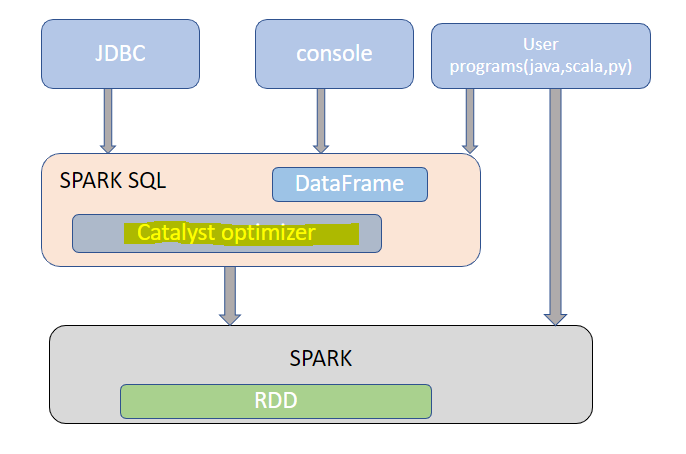
Here, we will learn about Spark SQL optimization – Spark catalyst optimizer framework. Optimization refers to a process in which we use fewer resources and jobs to run efficiently. We will discuss the role of catalyst optimizer in Spark. At last, we will also focus on its fundamentals of working and includes phases of Spark execution flow.

At the very core of Spark, SQL is a catalyst optimizer. It is based on a functional programming construct in Scala. Furthermore, the catalyst optimizer in Spark offers both rule-based and cost-based optimization as well. But, In rule-based optimization, there are rules to determine how to execute the query. While in cost-based by using rules, many plans are generated, and then their cost is computed.

In addition, there is a general library to show trees and apply rules to manipulate them. There are libraries specific to relational query processing, on top of this framework, for example, expressions, logical query plans.

It also attains the various set of rules for different phases of query execution such as analysis, logical optimization, physical planning, and code generation. These are used to compile parts of queries to Java bytecode.

Moreover, it uses another Scala feature, quasi quotes. That makes it easy to generate code at runtime from composable expressions. Ultimately, catalyst offers several public extension points, including external data sources & user-defined types.



**Role of Catalyst Optimizer in Apache Spark**

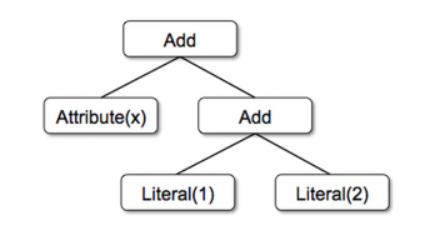
There are two purposes to design catalyst optimizer:

1. To add easily new optimization techniques and features to Spark SQL. To handle various problems going with significant data issues like semistructured data and advanced analytics.

2. To enable external developers to extend the optimizer.

**Fundamentals of Apache Spark Catalyst Optimizer**

**1. Trees:** Trees constitute the primary data type in a catalyst. Each node has a node and can have zero or more children. In Scala, new node types are explained as subclasses of the tree node class. Since these objects are immutable, by using functional transformations, we can manipulate them easily. By using a simple example, suppose for an elementary expression language, we have three node classes: Lit(value: Int): a constant value Att(name: String): an attribute from an input row, e.g., “x” Add1(left: TreeNode, right: TreeNode): sum of two expressions. To build up trees, we use all of these; for example, we can represent a tree, for an expression like x+(1+2) in Scala code as follows:



**2. Rules:**By using rules, we can manipulate trees. Those are functions from a tree to another tree. By using the rule, we can run arbitrary code on its input tree. Moreover, this approach uses a set of pattern matching functions that find and replace subtrees with a specific structure; trees offer a transform method in a catalyst. On all nodes of a tree, we apply the pattern matching method recursively. By transforming the ones, it matches each pattern to a result. By using this example, implementing a rule that folds add operations between constants:

tree.transform { case Add(Lit(c1), Lit(c2)) => Lit(c1+c2) }

Applying this to the tree for x+(1+2) would make the new tree x+3. In this expression, the case keyword is Scala’s standard pattern matching syntax; we use it to match the type of an object. It is also used to give names to extracted values (c1 and c2 here). An expression of pattern matching is passed to transform is a partial function. That means it only needs to match to a subset of all possible input trees.

Furthermore, to fully transform a tree, rules may need to execute multiple times. Catalyst groups rule into batches; it runs each set until it reaches a fixed point. In other words, until the tree stops changing after applying its rules. Ultimately, rule conditions contain arbitrary Scala code. It provides catalyst more power than domain-specific languages for optimizers while keeping it concise for simple rules.

**Spark Catalyst Optimizer**

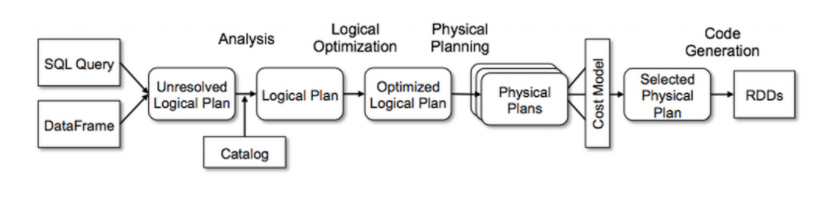
There are 4 phases in which we can use catalyst’s general tree transformation framework:

1. By analyzing a logical plan to resolve references.

2. With logical plan optimization.

3. By Physical Planning.

4. With code generation to compile parts of the query to Java bytecode.



1. Analysis

The first phase of Spark SQL optimization is analysis. Initially, Spark SQL starts with a relation to be computed. It can be calculated in two ways, either from an abstract syntax tree (AST) returned by a SQL parser. Using API, a second way is from a dataframe object constructed. Moreover, on applying any case, the relation remains unresolved attribute relations such as, in the SQL query SELECT col FROM sales, or even if it is a valid column name, is not known until we look up the table.

An attribute remains unresolved if we do not know its type to an input table. To resolve these, SQL uses catalyst rules that track tables in all data sources. It begins by building an “unresolved logical plan” tree with attributes & data types. Afterward, applies rules that do the following :

* – Searching relations as the name of the catalog.
* – Map the name attribute, for example, col, to the input provided given operator’s children
* – Determine which attributes match the same value to give them a unique ID.
* – By propagating and pushing types through expressions

2. Logical Optimization

It is the second phase of Spark SQL optimization. In this phase, we apply standard rule-based optimization to the logical plan. There are several rules in this process, for example, predicate pushdown, constant folding, and many other rules. Ultimately, it became elementary to add a rule for various situations.

3. Physical Planning

In physical planning rules, there are about 500 lines of code. From the logical plan, we can form one or more physical plans in this phase. Actually, by using the cost mode, it selects the plan. Only to select join algorithms, it uses cost-based optimization.

The framework supports the broader use of cost-based optimization, especially for all small relations. Moreover, By using this rule, it can estimate the cost recursively for the whole tree. Pipelining projections or filters operation, which is physical optimization, is also carried out. These projections are rule-based optimizations, and they can push operations from the logical plan into data sources. That supports predicate or projection pushdown.

4. Code Generation

The code generation is the final phase of SparkSQL optimization. To run on each machine, generation of Java bytecode is involved. Also, make code generation easier, as catalyst uses the unique Scala feature, “Quasiquotes.” Else, it is very tough to build code generation engines.

We can construct abstract syntax trees (ASTs) in Scala using Quasi-quotes. That can be given to the Scala compiler at runtime to generate bytecode. Furthermore, We can quickly transform a tree representing an expression in SQL to an AST. Especially for Scala code by using a catalyst. Also, it helps to evaluate that expression and then compile and run the generated code.

**Conclusion**

Hence, [Spark SQL](https://www.projectpro.io/project-use-case/spark-sql-on-spark-2) optimization enhances the productivity of developers and the performance of the queries that they write. A good query optimizer automatically rewrites relational queries to execute more efficiently, using techniques such as filtering data early, utilizing available indexes, and even ensuring different data sources are joined in the most efficient order.

By performing these transformations, the optimizer improves the execution times of relational queries and frees the developer from focusing on the semantics of their application instead of its performance. Catalyst uses Scala’s powerful features such as pattern matching and runtime metaprogramming to allow developers to specify complex relational optimizations concisely.

**Explain Resource Allocation configurations for a Spark application.**

Resource Allocation, i.e., Distribution of Executors, Cores, and Memory for a Spark Application, is an essential aspect during the execution of any spark application. If not configured correctly, a spark application may consume entire cluster resources and make other applications to be in the queue for a long time for resources. Here, we are trying to understand the basic flow in a Spark Application and then how to configure the number of executors, memory size determining each executor, and the number of cores for a Spark Application.

**Partitions:** A partition is a small chunk of a large distributed data set. Partitions are basic units of parallelism in [Apache Spark](https://www.projectpro.io/article/hadoop-mapreduce-vs-apache-spark-who-wins-the-battle/83). Spark manages data using partitions that help parallelize data processing across the cluster.

**Task:** A stage is generally a collection of Tasks. A task is a unit of work that can be run on a partition of a distributed dataset and gets executed on a single executor. All the tasks within a single stage can be executed in parallel. The unit of parallel execution is at the task level.

**Executor:** An executor is a single JVM process launched for an application on a worker node. Executor runs tasks and keeps data in memory. Each application has its executors. A single node can run multiple executors, and executors for an application can be run on multiple worker nodes. An executor lives for the duration of the execution of Spark Application and runs the tasks in numerous processes. The number of executors for a spark application can be specified inside the SparkConf or from the "spark-submit" by using -–num-executors.

**Cluster Manager:** It schedules and divides resources in the host machine which forms the cluster. The prime work of the cluster manager is to divide resources across applications. It works as an external service for acquiring resources on the cluster. Spark is agnostic to the underlying cluster manager. As long as it can acquire executor processes and communicate with each other, it is relatively easy to run it even on a cluster manager that also supports other applications (e.g., Mesos/YARN). Apache Spark system supports three types of cluster managers, namely- a) Standalone Cluster Manager, b) Hadoop YARN c) Apache Mesos.

**Yarn-client mode:** In client mode, the driver runs in the client process, and the application master is only used for requesting resources from YARN.

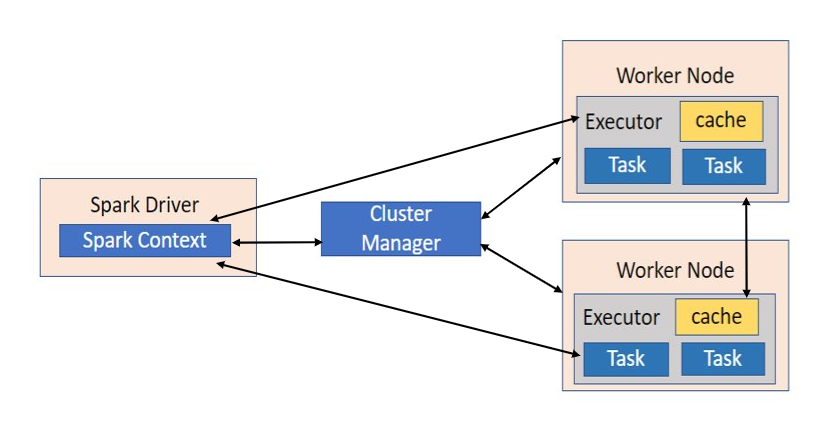
**Yarn-cluster mode:** In cluster mode, the Spark driver runs inside an application master process managed by YARN on the cluster, and the client can go away after initiating the application.

**Cores:** A core is a basic computation unit of a CPU, and a CPU may have one or more cores to perform tasks at a given time. The more cores we have, the more work we can do. In spark, this controls the number of parallel tasks an executor can run.

1. Spark Job Execution

Let's review the concepts of these two critical shared variables, but before that, here is a diagram showing how the Spark job is executed in the cluster.

Driver node submits Spark job to the cluster manager. First, it creates the spark context that coordinates with the cluster manager for cluster resources (worker nodes), and in response cluster manager allocates worker nodes from the cluster. Spark context or driver program then launches Executors on the worker nodes. Multiple executors can be launched on a single worker node. Each executor then launches numerous tasks that run concurrently. General practice shows that each executor should be configured to launch a maximum of 5 tasks concurrently; otherwise, contention between tasks will degrade the overall Spark job.



There are a few factors that we need to consider to decide the optimum numbers for the above three, like:

1. The amount of data to be processed
2. How quick a job has to complete?
3. whether using Static or dynamic allocation of resources
4. Upstream or downstream application

There are two ways in which we configure the executor and core details to the Spark job.

* **Static Allocation –** The values are given as part of spark-submit, such as --num-executors,--executor-cores,--executor-memory.
* **Dynamic Allocation –** which scales the number of executors registered with this application up and down based on the workload. The values are picked up based on the requirement (size of data, amount of computations needed) and released after use. This helps the resources to be re-used for other applications. By default, in spark, this is set to **False**. It can be enabled by setting **spark.dynamicAllocation.enabled** to **True**.

Usually, the Hardware configuration of Machines changes as per requirement. Here below, we will consider a case where we have some predetermined hardware config, and we need to determine the optimized resource allocation for an application.

**Case 1: Hardware – 6 Nodes, and each node have 16 cores, 64 GB RAM**

First, one core and 1 GB are needed for Operating System and Hadoop Daemons on each node, so we have **15 cores, 63 GB RAM for each node**.

**The number of cores:**

Number of cores = Concurrent tasks an executor can run

So we might think, more concurrent tasks for each executor will give better performance. But research shows that any application with more than 5 concurrent tasks would lead to a bad show. So, the **optimal value is 5**.

This number comes from an executor's ability to run parallel tasks and not from how many cores a system has. So, the number 5 stays the same even if we have double (32) cores in the CPU.

**The number of executors:**

From above, we determined **5 as cores per executor** and 15 as total available cores in one node (CPU) – we come to 3 executors per node, which is 15/5. We need to calculate the number of executors on each node and then get the total number for the job. So, the **number of executors per node is 3**.

So, with 6 nodes and 3 executors per node – we get a total of 18 executors. Out of 18, we need one executor (java process) for Application Master in YARN. So final number is 17 executors. So, this 17 is the number we give to spark using –num-executors while running from a spark-submit shell command.

**Memory for each executor:**

From the above step, we have three executors per node. And available RAM on each node is 63 GB. So memory for each executor in each node is 63/3 = 21GB. However, small overhead memory is also needed to determine the total memory request to YARN for each executor.

The formula for that overhead is max(384, .07 \* spark.executor.memory)

Calculating that overhead: 0.07 \* 21 (Here 21 is calculated as above 63/3) = 1.47 GB per executor.

Since 1.47 GB > 384 MB, the overhead is 1.47, we need to take the above from each 21 above => 21 – 1.47 ~ 19 GB. So **executor memory – 19 GB**.

Final numbers – **Executors – 17, Cores 5, Executor Memory – 19 GB**

**Case 2: Hardware – 4 Nodes, and Each node have 16 Cores, 32 GB**

It is a sample case we are calculating directly using the above notion.

The number of cores is 5 is the same for good concurrency as explained above.

Number of executors for each node = 32/5 ~ 3.

So total executors = 4 \* 3 Nodes = 12.

Then final number is 12 – 1(for Application Master) = 11 executors

Executor memory :3 executors for each node. 31/3 ~ 10.

Overhead is .07 \* 10 = 700 MB. So, rounding to 1GB as overhead, we get 10-1 = 9 GB

Final numbers – **Executors – 11, Cores 5, Executor Memory – 9 GB**.

**Case 3: When more memory is not required for the executors**

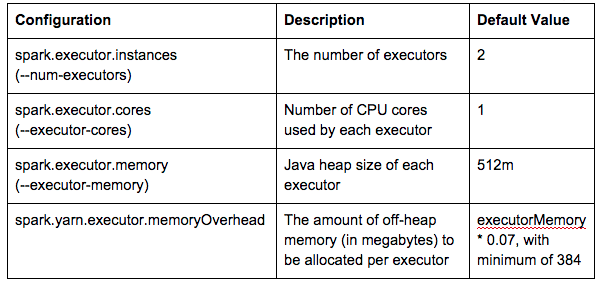
The above scenarios start with accepting the number of cores as fixed and moving to the number of executors and memory.

Now for the first case, if we think we do not need 19 GB, and just 10 GB is sufficient based on the data size and computations involved, then the following are the numbers of Cores: 5. The number of executors for each node = 3. Still, 15/5 as calculated above. At this stage, this would lead to 21 GB, and then 19 as per our first calculation.

But since we thought 10GB was ok (assume little overhead), we cannot switch the number of executors per node to 6 (like 63/10). Because with six executors per node and five cores, it comes down to 30 cores per node when we only have 16 cores. So we also need to change the number of cores for each executor.

So, calculating again. The number of cores from 5 comes to 3 (any number less than or equal to 5). So with three cores and 15 available cores – we get 5 executors per node, 29 executors (5\*6 -1), and memory is 63/5 ~ 12. Overhead is 12\*.07=.84. So, executor memory is 12 – 1 GB = 11 GB.

The final Numbers are **29 executors, three cores; executor memory is 11 GB.** After comparing to case1, this shows the number of cores and executor memory decreased, but the number of executors is increased.



**Spark Job Optimization Myth #1: Increasing the Memory Per Executor Always Improves Performance**

**Background**

One of the things I see most often when looking at client's clusters is a poor usage of their resources by their clusters. I see clients that have 100% memory usage but only 50% CPU usage complain about needing more nodes in order to run their jobs. Or there are others who have jobs that sit on the cluster for ages while they compute, taking up resources and slowing up everyone else. These sorts of issues can sometimes be managed through YARN queue management, but this ignores the root causes of the issues.

Additionally, I'm often told that Spark jobs are running slower than Hive jobs on their clusters, so they've never moved to Spark. While Hive jobs can run faster than Spark jobs, in most cases, Spark is the better option and should be comparable.

While these two issues seem to be separate, they often can be traced to the same root cause: a poorly optimized Spark job running on the cluster. For the new year, I'm going to be starting a series on myths that I've seen people believe when it comes to Spark job optimization. Some of these might be obvious to you, and others may come as a surprise, but I hope the information is useful.

**Spark on YARN Details**

One of the biggest issues I see is developers who just keep increasing memory whenever they run into issues. Not only is this haphazard and leads to inconsistent results, but it also doesn't actually do what they think it does.

To understand why it's important to go back to the basics. Spark only runs jobs when an action is called, which is an operation that requires instance feedback, such as collect or count. An action is made up of multiple stages, which represent operations leading up to that action from either the source of the data or cached results that were previously calculated. Stages are split based on necessary shuffles or movements of data, so multiple operations could be contained in a single stage. A stage is further split into tasks, each of which handles the operation for a subset of the data. This task is the base unit and the level at which Spark job optimization should start.

The graphic below shows how all of this works in a real-world, albeit simplified example. We have a simple program that is reading in data, grouping it by the tens and ones place, and then getting the count of each value. As you can see, the action here is the last line, which writes out to a CSV file. When that is executed, we first need to read the file and get the remainder, which makes the first stage. The group by requires a shuffle, so that is moved to a second stage, along with the writing of the data into HDFS.

**Myth #1: Increasing the Memory Per Executor Always Improves Performance**

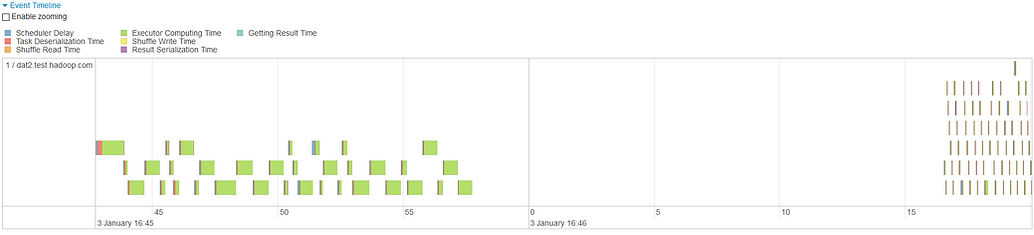
Getting back to the question at hand, an executor is what we are modifying memory for. A given executor will run one or more tasks at a time. Assuming a single executor core for now for simplicity's sake (more on that in a future post), then the executor memory is given completely to the task. In this instance, that means that increasing the executor memory increases the amount of memory available to the task.

Given this, it's pretty clear that increasing memory is not going to make a task run faster. And because this only impacts a single task (no cross-talk due to multiple executor cores or parallelism or anything like that), optimizing the memory for the task is of paramount importance, and will give us the result we want.

**What Then?**

So, if increasing memory isn't the right answer, what is? Well, there are two things we can improve: we can make more tasks run at the same time, and we can make the individual tasks run faster. Making the individual tasks run faster is not something that is easy to do when using the new Spark data frame API since that optimization is largely done for you automatically. Because of this, we'll look at making more tasks run at the same time.

First step to see if we can make more tasks run at the same time is to see if we are serializing tasks on the executor currently. In Spark History Server, you can see the timeline for a given stage. If it looks something like below, then you have tasks that are waiting for time on the executor. If we increase the number of executors, then we could have more tasks running at once.



Keep in mind that increasing the number of executors will increase the resource usage of your job, but in this case, it will also improve performance, so the trade-off can be worth it.

This obviously isn't the only way to tackle this issue, and we'll discuss other techniques in future posts.

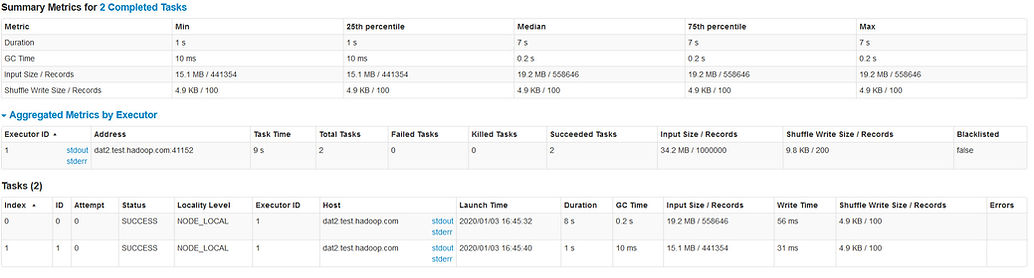
**Then How Much Memory Is Enough?**

Once we've seen how to improve performance, the next question is often how much executor memory we should give. This is a trickier question, and, as with most things in technology, the general answer is "it depends".

One place you can go to determine this is looking at the data you are processing. If you are processing 2 GB of data (not exactly big data, but it happens), then it's safe to assume 2 GB is a good maximum executor memory size. You may need more, but it should be rare.

Also, keep in mind that as long as you aren't doing joins or group bys, you can safely assume each executor will hold (Total Size of Data) / (Number of Tasks) GB of data. I wouldn't strictly stick to this calculation, however, and encourage up to double that value to give space for spikes in input data size. Joins and group bys introduce the possibility of skew or data from other data sources is included, so more memory will be necessary there too. One final thing to keep in mind here is that the number of tasks is determined in various different ways, depending on the source of the data frame. Additionally, you can use repartition to change the number of tasks, if necessary.

One final way you can determine whether your current memory allocation is too high or not is to look at the Spark History Server again. Looking there in the stage details page, you can find output that looks like the following:



As you can see here, we had two tasks run, each of which processed 15 - 20 MB of data. The input dataset I'm using is 34.2 MB, so this makes sense. Based on this, we can probably set the executor memory to 128 MB and still be an order of magnitude safe from any spikes we might see. Doing this will not make performance worse at all, and in fact, may make the performance slightly better due to garbage collection times and wait times on the YARN queue.

**Spark Job Optimization Myth #2: Increasing the Number of Executors Always Improves Performance**

Last week, I introduced a series I am going to be doing focusing on myths I see often dealing with Apache Spark job optimization. [My first post](https://www.davidmcginnis.net/post/spark-job-optimization-myth-1-increasing-the-memory-per-executor-always-improves-performance) focused on the common reaction I see for developers to add memory when they need to speed up their jobs. We discussed why this doesn't always work, and what to do instead. One aspect of that solution was to instead increase the number of executors, which can give you better results in some situations.

This week I'm going to turn that on its head, and look at why increasing the number of executors doesn't always work. Increasing the number of executors certainly works more often than increasing the amount of memory per executor, but there are still very important times when it doesn't work the way you'd want it to.

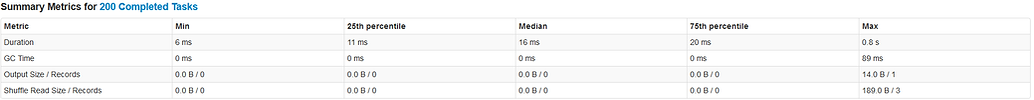
**I Thought We Were Supposed to Increase the Number of Executors?**

If you read the first post last week, one of my final recommendations was to increase the number of executors, which can often work. You may have tried this on your jobs, and it still didn't improve the performance. So did I give bad advice last week, or is your job just as fast as it can possibly be, or is there something else?

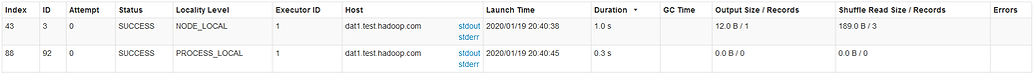
I definitely didn't give bad advice last week, and dozens of jobs I've optimized in that manner speak to that. Your job might be as optimized as it can be, but there are many other things it could be as well. I'm going to go through a few possibilities that may cause this, and how to detect and handle these scenarios. These don't cover every possible case, but they should give you a good idea of the possibilities.

**Data Skew**

Data skew is common in Spark jobs, and can often happen in situations you wouldn't expect it. Whenever I find that a job isn't running as fast as I'd expect, one of the first things I check is data skew. I've handled this topic in-depth in a [previous post](https://www.davidmcginnis.net/post/spark-job-optimization-dealing-with-data-skew), so I'll just mention how to detect skew here. If you find you have skew, that post does a great job of going through your options.



The screenshot above is from the Spark History Server's details for a specific stage. In this screenshot, you can see the skew pretty easily. The bottom 75% of jobs took 20 milliseconds to run, yet the maximum job took 800 milliseconds. If we dive into the actual executor times individually and sort by the runtime, we find that the maximum run time (which here is confusingly labeled as having taken 1 second) is more than three times longer than the next longest task.



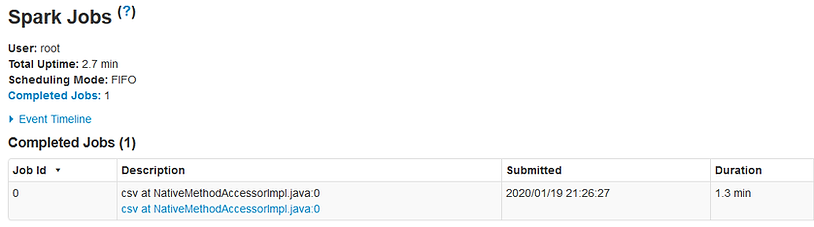
This is a textbook example of skew. While all skewed jobs may not be as obvious, the same techniques should show them to you. Thankfully this is made pretty easy by the Spark History Server.

**Long-Running Driver Code**

Another common scenario I have come across is when developers misunderstand the use of Spark and end up writing most of their code to run on the driver. This issue is closely related to the over-use of collect since oftentimes this is caused by collecting data locally in order to process, before sending back up to the cluster. It goes without saying that if you are doing this, either you can hold everything in memory, in which case you should consider whether Spark is the right move for you, or you can't, and you're going to fail.

In these situations, there are many ways to detect how long the driver is running. The first and most obvious is looking at your logging infrastructure for the job. If you see a lot of custom logs in the driver code, then you're likely using the driver more than you should. You can even use the timestamps to get a feel for exactly how long the driver is taking.

If you need more exact than that, the Spark History Server can help. On the main page for the application, it gives a length of all of the jobs, and also a length for the overall application. Comparing these two gives you an idea of how much time was spent outside of actually doing work. There is always going to be some overhead on jobs (on my small cluster for simple applications, I saw around 50 - 60 seconds of overhead on average), but if it gets to be too much, that's a warning flag. See the example below, that has an 84-second difference between application runtime and job runtime. This is much more than the typical 50 - 60 seconds I'm seeing for most jobs on this particular cluster, so it warrants further investigation.

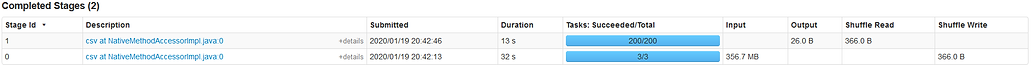


How we fix these issues depends heavily on your driver. Most often this indicates that your architecture is off, and you're using the driver more than you should. Sometimes this is expected and you just need to live with it or do some optimization on the driver code itself.

**Not Enough Tasks**

This is probably the closest you'll get to a job that is unable to be optimized further without actually reaching that point. In this scenario, your number of tasks is less than the number of executors you have set up. This can be the number of tasks for a specific stage you are trying to optimize, or it can be the maximum number of tasks used in all stages in the job.

This is easily detected in Spark History Server as well, by comparing the number of tasks for a given stage to the number of executors you've requested. The easiest way to see how many tasks per stage is in the job details page, where it shows the progress bar of tasks completed, as seen below.



You can see the first stage for this job only uses 3 tasks. That means that there is no way that increasing the number of executors larger than 3 will ever improve the performance of this stage. The second stage, however, does use 200 tasks, so we could increase the number of tasks up to 200 and improve the overall runtime.

Based on the fact that the stage we can optimize is already much faster than the other stage, however, increasing the number of tasks further likely won't yield enough of a benefit to be worth it. Remember to always keep the balance in mind between using more resources and the benefit you get from it, to avoid wasting resources. If we were just barely under SLA currently, adding more executors could get us under with some room to spare. But it is physically impossible to get more than 13 seconds out of that, based on the above run, and likely that number is closer to 5 seconds.

**Conclusion**

As I stated last week, increasing the number of executors often times will improve your performance. You shouldn't just blindly do this, however, and hopefully, the scenarios above give you some pause before you do that next time. These scenarios are not the only ones possible, but hopefully, they demonstrate the variety of ways it can happen and give you ideas for what to look for as you work on optimizing your jobs.

**Spark Job Optimization Myth #3: I Need More Driver Memory**

For the last few weeks, I've been diving into various Spark job optimization myths which I've seen as a consultant at my various clients. I started with [why increasing the executor memory may not give you the performance boost you expect.](https://www.davidmcginnis.net/post/spark-job-optimization-myth-1-increasing-the-memory-per-executor-always-improves-performance) Last week we discussed [why increasing the number of executors also may not give you the boost you expect.](https://www.davidmcginnis.net/post/spark-job-optimization-myth-2-increasing-the-number-of-executors-always-improves-performance)

This week, we are going to change gears a bit, and focus on the driver. While the driver is not oftentimes the source of a lot of our issues, it is still a good place to look at for improvements. Every little improvement adds up, after all.

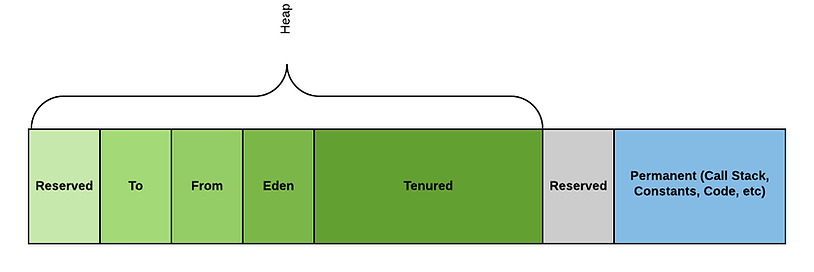
The most common misconception I see developers fall into with regards to the driver configuration is increasing driver memory. We'll discuss why this is generally not a good decision, and the rare cases when it might be reasonable to do.

As with previous weeks, I'm running tests on a local 3-node HDP 2.6.1 cluster using YARN. I've set YARN to have 6 GB total and 4 cores, but all other configuration settings are default. I haven't enabled security of any kind. I'm using Vagrant to set this up, and have supplied the Vagrant file on [Github](https://github.com/davidov541/HadoopOnVagrant/tree/master/HDP2.6.X-CentOS7).

**What Does the Driver Look Like Anyways?**

Oftentimes when writing Spark jobs, we spend so much time focusing on the executors or on the data that we forget what the driver even does and how it does it. This actually isn't a horrible thing, however, since, from its view, it is just any other Java/Scala/Python/R program, using a library called Spark. There's no fancy memory allocation happening on the driver, like what we see in the executor, and you can even run a Spark job just like you would any other JVM job, and it'll work fine if you develop it right.

Based on this, a Spark driver will have the memory set up like any other JVM application, as shown below. There is a heap to the left, with varying generations managed by the garbage collector. This portion may vary wildly depending on your exact version and implementation of Java, as well as which garbage collection algorithm you use. The right-hand side is your permanents, where things like the stack, constants, and the code itself are held. This should all be very familiar to you if you've ever taken a computer architecture course. If not, you don't need to understand the details here, just that it is similar to any other JVM application.



Keep in mind in all of this there are a few exceptions that we are glossing over for simplicity's sake. One example is if you use YARN cluster mode, then YARN will set up your JVM instance for you, and do some memory management, including setting the heap size for you. For the most part, these are transparent and don't hae a huge effect on our results.

**So, What Does That Mean?**

At this point, unless you're a theoretical computer science junkie like me, you're probably asking yourself "so what?" and figuring out how much further you have to go. It's simple: optimize the driver code like you would optimize any Java application. This includes simple things like:

* Avoid unnecessary memory usage
* Only set the heap to what you need
* Don't use globals

Obviously, there are a lot more, but these three translate very well to Spark specific projects as well, so we'll focus on them. We'll talk about each of these as they pertain to Spark below.

**Avoid Unnecessary Memory Usage**

This is a pretty obvious one in a normal JVM application. If you don't need the contents of a huge file, don't read it in. Simple, right?

Yet so often I see applications that collect all of the data from a DataFrame into memory. This does exactly the same thing: takes a large amount of data stored safely elsewhere, and pulls it into memory. This is often done as a collect() call. Because of this, using collect() is often the first sign that something is wrong, and needs to be fixed.

Why collect your data to the driver to process it, when that is what Spark is there for? Collect calls should be used only when either you are in a development environment testing your code, or when you know 100% without a doubt that it will never be large. Even then, the second one is doubtful. After all, how many of us know 100% something will never happen in our applications?

**Only Set the Heap to What You Need**

Another one is not setting the heap size to be too large. Most generic JVM applications I've seen, the heap size isn't set unless it was found to be absolutely necessary. Sometimes that is in place of optimization, and sometimes that is despite optimization. Regardless, because it isn't too easy to set the heap size, most developers don't mess with it until they need to.

That's not the case in Spark. Spark makes it really easy, especially if you are using the YARN cluster mode. It's just another switch of the many you need to set anyways, so many people set it. Additionally, because there is the misconception that increasing executor memory speeds things up, that naturally translates to driver memory as well.

This is all wrong. The default driver memory size is 1 GB, and in my experience, that is all you need. Looking at the memory layout above, what do you expect to take up more than 1 GB of memory? The stack and constants should be small. The heap will have pointers to DataFrames, and maybe a configuration file loaded, but not much else. You shouldn't be collecting data, so that shouldn't be on the heap. There's just not much you need to have on the heap in a well-written Spark driver.

There is one caveat to this: [SPARK-17556](https://issues.apache.org/jira/browse/SPARK-17556). This is a known bug where if you use a broadcast join, the broadcast table is kept in driver memory before broadcasting it. This means that if you are using broadcast joins a lot, you are essentially collecting each of those tables into memory. In this case, it is reasonable to increase the memory usage for driver memory, until the bug is fixed.

**Don't Use Globals**

One final thing that you should avoid is globals. Global variables are bad for so many reasons, but one is that the data is kept around forever, even when it isn't needed anymore. So if you have a DataFrame that reads the data in from a file, but only need it once to start the processing, why keep it around? Yet, if it is kept in a global variable, it will be kept around for the entire application. Instead, put it in an object that will be removed once it can no longer be referenced due to leaving that scope. This saves you room and headaches down the road.

**Conclusion**

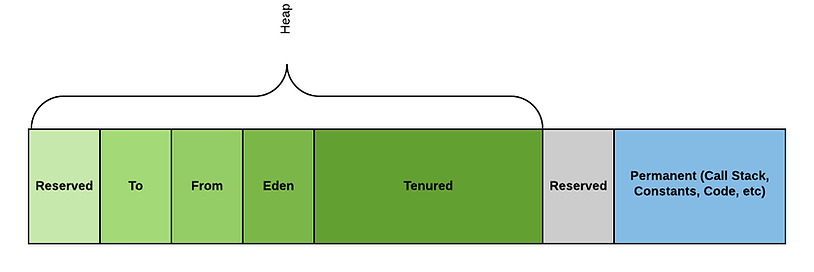
This isn't one of the flashiest optimizations you can do or one that will change your life, but it's an important optimization to run. Reducing your memory usage on the driver will lower your YARN usage amount and might even speed up your application.

Additionally, I've found that applying these patterns helps clear up the code immensely. It can be really difficult sometimes to determine where code is supposed to be running between the driver and executor. Enforcing these policies oftentimes will make that distinction clearer, making your application more maintainable.

**Spark Job Optimization Myth #4: I Need More Overhead Memory**

While I've seen this applied less commonly than other myths we've talked about, it is a dangerous myth that can easily eat away your cluster resources without any real benefit. Understanding what this value represents and when it should be set manually is important for any Spark developer hoping to do optimization.

What Is Overhead Memory?



The first question we need to answer is what overhead memory is in the first place. Overhead memory is essentially all memory which is not heap memory. This includes things such as the following:

* Call stacks
* Memory-mapped files
* Shared libraries
* Constants defined in Code
* The code itself

Looking at this list, there isn't a lot of space needed. Files and libraries are really the only large pieces here, but otherwise, we are not talking a lot of room.

The developers of Spark agree, with a default value of 10% of your total memory size, with a minimum size of 384 MB. This means that not setting this value is often perfectly reasonable since it will still give you a result that makes sense in most cases.

**Why Do People Increase It?**

The most common reason I see developers increasing this value is in response to an error like the following.

This error very obviously tells you to increase memory overhead, so why shouldn't we? Because there are a lot of interconnected issues at play here that first need to be understood, as we discussed above.

While you'd expect the error to only show up when overhead memory was exhausted, I've found it happens in other cases as well. This leads me to believe it is not exclusively due to running out of off-heap memory. Because of this, we need to figure out why we are seeing this.

**How Do We Solve The Error Instead?**

If we see this issue pop up consistently every time, then it is very possible this is an issue with not having enough overhead memory. If this is the case, consider what is special about your job which would cause this. The defaults should work 90% of the time, but if you are using large libraries outside of the normal ones, or memory-mapping a large file, then you may need to tweak the value.

Another common scenario I see is users who have a large value for executor or driver core count. Each executor core is a separate thread and thus will have a separate call stack and copy of various other pieces of data. Consider whether you actually need that many cores, or if you can achieve the same performance with fewer cores, less executor memory, and more executors. We'll be discussing this in detail in a future post.

If you look at the types of data that are kept in overhead, we can clearly see most of them will not change on different runs of the same application with the same configuration. Based on that, if we are seeing this happen intermittently, we can safely assume the issue isn't strictly due to memory overhead.

If the error comes from an executor, we should verify that we have enough memory on the executor for the data it needs to process. It might be worth adding more partitions or increasing executor memory.

If it comes from a driver intermittently, this is a harder issue to debug. The first check should be that no data of unknown size is being collected. If so, it is possible that that data is occasionally too large, causing this issue. Collecting data from Spark is almost always a bad idea, and this is one instance of that.

Additionally, you should verify that the driver cores are set to one. Setting it to more than one only helps when you have a multi-threaded application. Since you are using the executors as your "threads", there is very rarely a need for multiple threads on the drivers, so there's very rarely a need for multiple cores for the driver.

You may also want to understand why this is happening on the driver. Looking at what code is running on the driver and the memory that is required is useful.

One thing you might want to keep in mind is that creating lots of data frames can use up your driver memory quickly without thinking of it. An example of this is below, which can easily cause your driver to run out of memory.

Keep in mind that with each call to *withColumn*, a new dataframe is made, which is not gotten rid of until the last action on any derived dataframe is run. That means that if *len(columns)* is 100, then you will have at least 100 dataframes in driver memory by the time you get to the *count()* call.

If none of the above did the trick, then an increase in driver memory may be necessary. This will increase the *total memory*\* as well as the overhead memory, so in either case, you are covered. Increase the value slowly and experiment until you get a value that eliminates the failures.

**When Is It Reasonable To Increase Overhead Memory?**

The last few paragraphs may make it sound like overhead memory should never be increased. If that were the case, then the Spark developers would never have made it configurable, right? So let's discuss what situations it does make sense.

One common case is if you are using lots of execution cores. We'll discuss next week about when this makes sense, but if you've already made that decision, and are running into this issue, it could make sense. As discussed above, increasing executor cores increases overhead memory usage, since you need to replicate data for each thread to control. Additionally, it might mean some things need to be brought into overhead memory in order to be shared between threads. For a small number of cores, no change should be necessary. But if you have four or more executor cores, and are seeing these issues, it may be worth considering.

Another case is using large libraries or memory-mapped files. If you are using either of these, then all of that data is stored in overhead memory, so you'll need to make sure you have enough room for them.

In either case, make sure that you adjust your overall memory value as well so that you're not stealing memory from your heap to help your overhead memory. Doing this just leads to issues with your heap memory later.

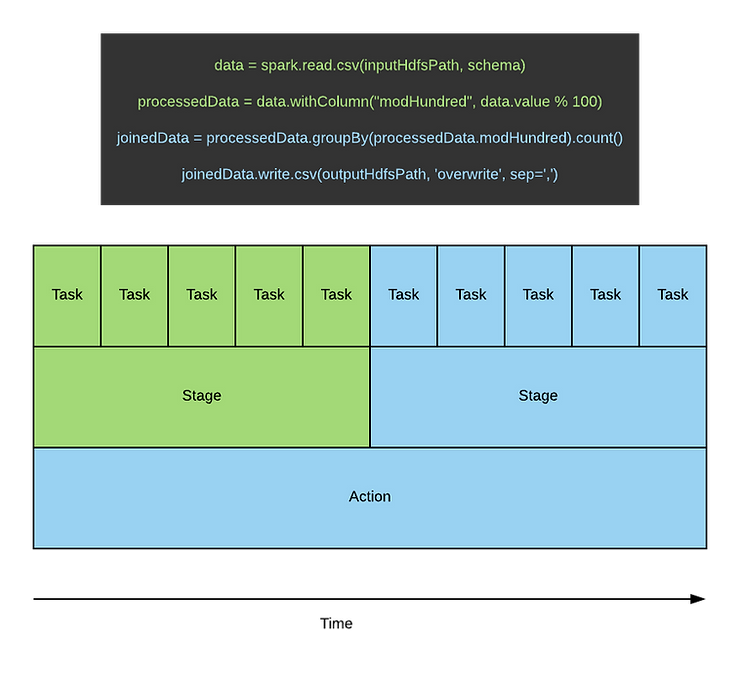
**Spark Job Optimization Myth #5: Increasing Executor Cores is Always a Good Idea**

This week, we're going to talk about executor cores. First, as we've done with the previous posts, we'll understand how setting executor cores affects how our jobs run. We'll then discuss the issues I've seen with doing this, as well as the possible benefits in doing this.

This is a topic where I tend to differ with the overall Spark community, so if you disagree, feel free to comment on this post to start a conversation. As always, the better everyone understands how things work under the hood, the better we can come to agreement on these sorts of situations.

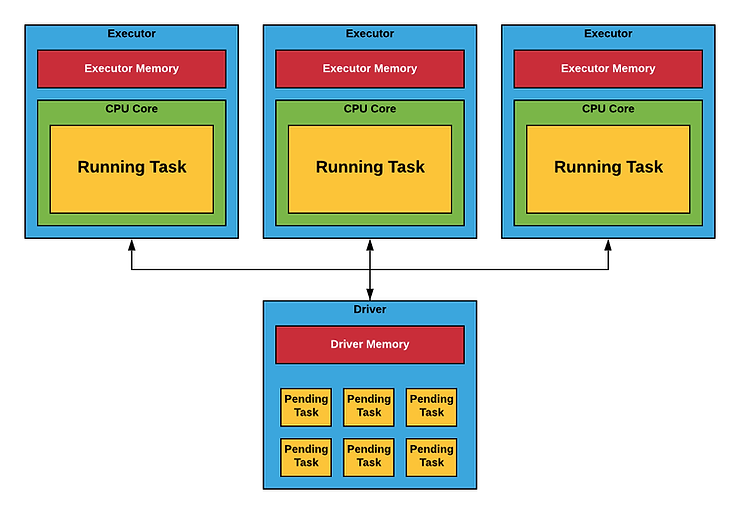
**How Does Spark Use Multiple Executor Cores?**

So the first thing to understand with executor cores is what exactly does having multiple executor cores buy you? To answer this, lets go all the way back to a diagram we discussed in the first post in this series.



As we discussed back then, every job is made up of one or more actions, which are further split into stages. These stages, in order to parallelize the job, is then split into tasks, which are spread across the cluster. Each task handles a subset of the data, and can be done in parallel to each other.

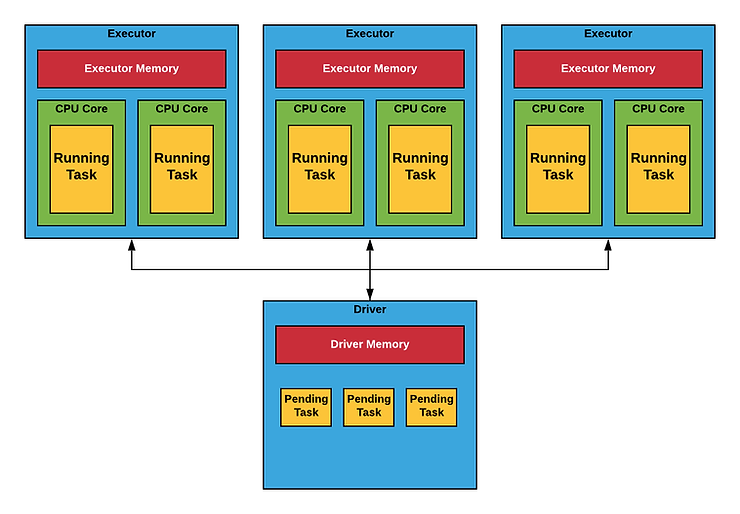
So how are these tasks actually run? Well if we assume the simpler single executor core example, it'll look like below. In this case, one or more tasks are run on each executor sequentially. As an executor finishes a task, it pulls the next one to do off the driver, and starts work on it. This allows as many executors as possible to be running for the entirety of the stage (and therefore the job), since slower executors will just perform fewer tasks than faster executors.



Note that we are skimming over some complications in the diagram above. Namely, the executors can be on the same nodes or different nodes from each other. Additionally, each executor is a YARN container. The driver may also be a YARN container, if the job is run in YARN cluster mode. Finally, the pending tasks on the driver would be stored in the driver memory section, but for clarity it has been called out separately.

So far so good. Now what happens when we request two executor cores instead of one? From the YARN point of view, we are just asking for more resources, so each executor now has two cores. Because YARN separates cores from memory, the memory amount is kept constant (assuming that no configuration changes were made other than increasing the number of executor cores).

Instead, what Spark does is it uses the extra core to spawn an extra thread. This extra thread can then do a second task concurrently, theoretically doubling our throughput. The result looks like below.



This seems like a win, right? We're using more cores to double our throughput, while keeping memory usage steady. Given that most clusters have higher usage percentages of memory than cores, this seems like an obvious win. Sadly, it isn't as simple as that.

**What's the Problem?**

Looking at the previous posts in this series, you'll come to the realization that the most common problem teams run into is setting executor memory correctly to not waste resources, while keeping their jobs running successfully and efficiently.

Let's say that we have optimized the executor memory setting so we have enough that it'll run successfully nearly every time, without wasting resources. Now let's take that job, and have the same memory amount be used for two tasks instead of one. It's pretty obvious you're likely to have issues doing that. That's because you've got the memory amount to the lowest it can be while still being safe, and now you're splitting that between two concurrent tasks.

This is essentially what we have when we increase the executor cores. Increasing executor cores alone doesn't change the memory amount, so you'll now have two cores for the same amount of memory. So once you increase executor cores, you'll likely need to increase executor memory as well. The naive approach would be to double the executor memory as well, so now you, on average, have the same amount of executor memory per core as before.

One note I should make here: I note this as the naive solution because it's not 100% true. Some memory is shared between the tasks, such as libraries. Assuming you'll need double the memory and then cautiously decreasing the amount is your best bet to ensure you don't have issues pop up later once you get to production.

And with that you've got a configuration which now works, except with two executor cores instead of one. But what has that really bought us now? We are using double the memory, so we aren't saving memory. At this point, we might as well have doubled the number of executors, and we'd be using the same resource count.

Increasing number of executors (instead of cores) would even make scheduling easier, since we wouldn't require the two cores to be on the same node. This means that using more than one executor core could even lead us to be stuck in the pending state longer on busy clusters.

Based on this, my advice has always been to use one executor core configurations unless there is a legitimate need to have more. But, this is against the common practice, so it's important to understand the benefits that multiple executor cores have that increasing the number of executors alone don't.

**Hidden Benefits**

The biggest benefit I've seen mentioned that isn't obvious from above is when you shuffle. If you shuffle between two tasks on the same executor, then the data doesn't even need to move. The data is still in the container in memory (or on disk, based on caching), so no network traffic is needed for that. This decreases your traffic utilization, and can make the network transfers that do need to occur faster, since the network isn't as busy.

Based on this, if you have a shuffle heavy load (joining many tables together, for instance), then using multiple executor cores may give you performance benefits. It is best to test this to get empirical results before going this way, however. The typical recommendations I've seen for executor core count fluctuates between 3 - 5 executor cores, so I would try that as a starting point. Keep in mind that you will likely need to increase executor memory by the same factor, in order to prevent Out of Memory exceptions. If you don't do this and it is still successful, you either have failures in your future, or you have been wasting YARN resources.

**Conclusion**

Based on the above, my complete recommendation is to default to a single executor core, increasing that value if you find the majority of your time is spent joining many different tables together. When doing this, make sure to empirically check your change, and make sure you are seeing a benefit worthy of the inherent risks of increasing your executor core count.

As I stated at the beginning, this is a contentious topic, and I could very well be wrong with this recommendation.

That said, based on my experience in recommending this to multiple clients, I have yet to have any issues. In contrast, I have had multiple instances of issues being solved by moving to a single executor core. I actually plan to discuss one such issue as a separate post sometime in the next month or two.

**Memory Exceptions, So I Need to Increase Memory**

In this entry, we're going to discuss something a bit different, and that is the common excuse I see from developers to immediately try to increase resources, instead of understanding the system they are building. And that excuse is the notorious *OutOfMemoryError* exception.

This error can lead to many different solutions based on a variety of factors, including where the error came from, what other errors are seen on that node and across the system, and the exact message included with the exception. Because of that, we're going to focus on what you should do when you see this exception, so that you'll better understand what the error is actually trying to tell you before you waste cluster resources.

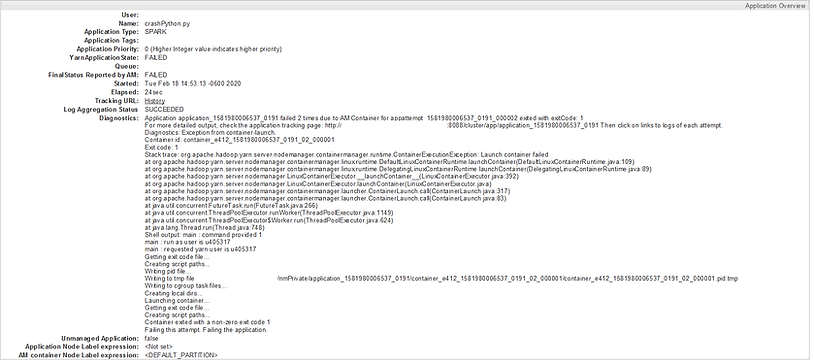
**Determining the True Source of the Error**

The first step when you see this error is to figure out where the error is occurring. Is it occurring on the driver? Or is it on only one of the executors? Or maybe all of the executors are seeing this error? If you are running in YARN client mode, then any errors from the driver will be printed to standard out. If this error is not printed to standard out in these cases, then the driver is not the one throwing this error.

If not, or if you are running in any other mode, then you'll need to look at the individual logs for the nodes. As we've discussed in past posts, using the *yarn logs* command will help you immensely here, along with a good helping of *grep* magic. You can use the exact error message from the original issue you saw to find the error messages quickly across all nodes. Don't use the timestamp, since this will preclude other executors that may have run into the same issue at different times.

One note I want to make here is that I often see developers pointing to the YARN resource manager UI for their job like the image below, and do all of their debugging based on this. Note that while this one doesn't mention memory issues specifically, there are definitely situations where it will show an OutOfMemoryError exception in this output.

The error shown here is just a single error in what may be many different, interconnected (or not) errors. This gives you a good idea of what to look for in the logs to find the actual error, but it isn't going to be all you need.



**Is This the Error We're Looking For?**

Once you find where the error is coming from, the next step we need to take is understand what other errors and warnings were being reported around the same time on the same nodes. If the error was reported on the driver, understanding any errors in the executors will also be necessary. If the error happens when interfacing with an external system (such as a JDBC connection), then check the logs on that system to make sure an issue didn't happen there.

This may all seem to be overkill. We have the error, right? Well, not quite. It's rare for a job to have one error reported and that be the error we are looking for. Instead, it's common to see multiple errors across multiple nodes, all interconnected. As an example, you'll often see network errors on the driver when an executor crashes. The issue isn't the network errors, but instead whatever caused the executor to crash. Yet both will be reported.

Checking for errors and warnings (using the ERROR and WARN keywords from log4j) gives us an understanding of all of the errors being reported. Keep the timestamps in mind as well, since often the earliest errors are the root issues.

**When It Really Is a Memory Issue**

If you go through the steps above, and are left with a memory issue as the most likely culprit, there's still some more work to be done. If the error is coming from the driver, then looking at the possible causes in my [earlier post in this series on driver memory](https://www.davidmcginnis.net/post/spark-job-optimization-myth-3-i-need-more-driver-memory) would help. If it's coming from only one or two executors, then you might be [dealing with skew](https://www.davidmcginnis.net/post/spark-job-optimization-dealing-with-data-skew). Finally, the cause for the OOM exception may indicate a specific set of memory such as [overhead](https://www.davidmcginnis.net/post/spark-job-optimization-myth-4-i-need-more-overhead-memory) or permanent generation. These are handled in specific ways and may be caused by specific things, so you'll want to dig into exactly what that type of memory is used for, and what you can do to reduce that usage.

**Conclusion**

Increasing memory usage by your application may not be the end of the world, but jumping to that solution is a common practice among developers that needs to change in order to build better applications. Follow the steps above, and you'll be many steps ahead of your peers in writing better, more efficient code.

And with that, I'm ending our series on Spark job optimization myths. I hope that you've gained a lot of good information from this series, and use it to improve your jobs. Feel free to comment here, or tweet me any other myths or topics you'd like me to cover, and I'll consider them for a future post.

**Spark Job Optimization: Dealing with Data Skew**

**Spark Job Optimization**

Optimizing a Spark job can be a daunting task. You have a job that you've functionally completed, but you're running into any number of random exceptions. To make matters worse, those exceptions keep changing or don't match your expectations. This is because of many different reasons, including that it is easy to write a Spark job without really understanding how Spark truly works.

To that end, I'm going to be starting an intermittent continuing series on optimizing Spark jobs. This series is going to focus on diving into the inner works of Spark, helping you optimize your jobs by better understanding your jobs.

**Data Skew**

The first topic we're going to tackle is data skew. Data skew happens when for one reason or another, a small percentage of partitions get most of the data being processed. In normal usage, Spark will generally make sure that the data is evenly split across all tasks, so there isn't a big risk of skew. When you do a join, however, Spark distributes the data by join key, so that data from the two tables being joined will be in the same task. If you have a lot of rows with the same key, then you have some tasks with those keys taking much longer than the others.

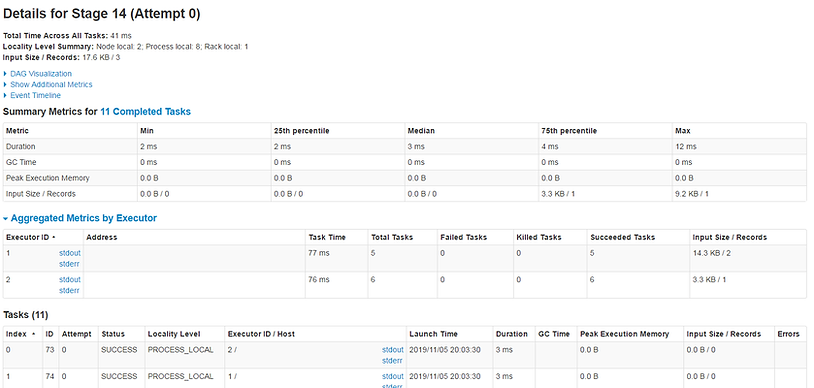
Overall, data skew generally will lead to one of two situations.

The first is that the job will just plain fail, due to workers running out of memory. This may show up as an OutOfMemoryException or a similar error. Often, this will lead to developers increasing the executor memory limit until everything passes. While this does get the job running, it uses much more resources than necessary. It also leads to the second situation.

The second situation that can happen is when jobs take much longer than they should. While this isn't as catastrophic as the first situation, and often is overlooked, it is still an issue. This situation can be a difficult one to detect, but looking at the History Server entry will help here.

Before we get to that, we need to discuss a bit about how Spark works under the hood. Data in Spark is split into individual partitions. Any time that a Spark job calls an action, or a function which requires some execution to occur, a stage is created to perform that action. Examples of actions include collect, show, and count. A stage in turn has a child task for each partition, which is then run on exactly one executor (assuming no failed tasks or speculative execution).

In the Spark History Server, look through the task list for stages that are taking the longest. Once you click on a stage, you'll see a page similar to the one below.



It is fairly obvious at first glance that this job is very small, but for a larger job that needs optimization, the same values will be visible. At the top, you can see the minimum, maximum, and various percentile values for different metrics. Note that in the "Show Additional Metrics" section I've selected Peak Execution Memory. Skew will be visible here when duration or peak execution memory is bottom-heavy. That is to say, the difference between the minimum and 75th percentile value is small compared to the difference between the 75th percentile and the maximum value. This means that there were a few tasks that, in turn, took much longer than others for this stage, which is indicative of skew.

For more verification of this, you can look at the individual tasks at the bottom, sorting by duration. This allows you to see the specific tasks that took a long time, and check their logs to ensure there weren't any extenuating circumstances causing the issue, instead of skew.

**Broadcast Hints**

So how do we solve this issue, now that we've identified it? The easiest way is to tell Spark to broadcast the table that has skew. If the table with skew also happens to be small, we can distribute that data to all of the tasks, so that there is no skew anymore. This is often most useful when you have a slow job that has a few long-running tasks, but memory usage is not a huge issue.

To use a broadcast hint, you can use either Spark SQL or normal code. The SQL code and Scala code look like the following

Broadcasts may be done automatically as well, but only if statistics are available for the data. If they aren't available currently, running a query similar to the following will compute the statistics on the Hive table, allowing broadcasts to be done automatically. For best results, this query needs to be run regularly, preferably immediately after significant changes to the table such as inserts.

One final note: due to [SPARK-17556](https://issues.apache.org/jira/browse/SPARK-17556), whenever you do a broadcast, the data is pulled onto the driver before pushing it out to all of the worker nodes. This can be surprising the first time, especially if you are diligent about keeping the driver memory usage low. Because of this, if you see driver side OOM exceptions, then it may be reasonable to move your driver memory to the same as your executor memory so that anything broadcast to your executors can fit on your drivers as well.

More information can be found [here](https://spark.apache.org/docs/latest/sql-performance-tuning.html#broadcast-hint-for-sql-queries). Note that the broadcast hint is just that: a hint. That means that Spark can ignore it, and actually will ignore it if the table is too large.

**Skew Hints**

Similarly, Spark also has a skew hint as well. This allows Spark to do various optimizations based on the assumption that a key is skewed. Note that this is a feature specific to Databricks, so it is not available on HDP or CDH. Because of that, I'm not going to spend much more time on it, but you can check out more information about it [here](https://docs.databricks.com/delta/join-performance/skew-join.html).

**Randomizing the Join Key**

Now we start getting into some of the less conventional methods for dealing with skew. The first one is randomizing the join key on the skewed table. This is based on an implementation by the [Data-R-Us Blog](https://datarus.wordpress.com/2015/05/04/fighting-the-skew-in-spark/). This method assumes that the skewed table is the larger table of the two, but the other table is too large to broadcast.

The first step is to modify the keys on the skewed table with a random seed. For example, if the key is a string, you might add "-N" at the end of each key, where N is a random number between 0 and 9 (or some other range, based on how big your dataset is, and how many partitions per key you want to create). In the other table, you then duplicate the data N times, each copy with the key modified to have a different "-N" appended. So for example, if you chose N to be between 0 and 2, then you would replicate the smaller table 3 times, one with the keys + "-0", one with the keys + "-1", and one with the keys + "-2". This way, the keys in the skewed table will match up with the smaller table.

You can then join the two tables together as normal. You would then need to modify the key to remove the additions, to get the same data back you had before.

This will cause have smaller partitions, so the skew will not affect the time and memory usage as much. Despite this, it does increase the data size. This is something to consider when choosing a strategy for dealing with skew.

**Custom Partitioner**

The last approach I'm going to mention here is a crazy yet effective one posted by [Sim at Swoop](https://stackoverflow.com/questions/38670369/what-is-an-efficient-way-to-partition-by-column-but-maintain-a-fixed-partition-c/38680666#38680666). The approach here is similar to the random approach above, but is more transparent to users, allowing it to be added as a library that is reused across teams.

In this approach, you create a custom partitioner that will purposefully distribute the key that is skewed to multiple hosts. You then distribute the corresponding key on the other table to all of them, to ensure that they can be joined properly.

This is not an approach for the weak, but if you can implement it well, it can work across many teams.

**Conclusion**

Skew can make a good Spark job work poorly. It takes up resources you don't need to use and makes your jobs run slowly at the same time. A lot of keys are naturally skewed, so this issue comes up pretty often. Hopefully, the solutions above help you reduce the amount of skew in your system, or at least put you on the right path. If you have another approach to reducing skew, feel free to add it to the comments below or send me a message!