[Big Data](http://site.clairvoyantsoft.com/category/big-data/), [CDH](http://site.clairvoyantsoft.com/category/cloudera/cdh/), [Cloudera](http://site.clairvoyantsoft.com/category/cloudera/), [Hadoop](http://site.clairvoyantsoft.com/category/hadoop/), [Software Development Methodologies](http://site.clairvoyantsoft.com/category/software-development-methodologies/), [Spark](http://site.clairvoyantsoft.com/category/hadoop/spark/)

Understanding Resource Allocation configurations for a Spark application

Posted on December 11th, 2016 by Ramprasad Pedapatnam

|

Resource Allocation is an important aspect during the execution of any spark job. If not configured correctly, a spark job can consume entire cluster resources and make other applications starve for resources.

This blog helps to understand the basic flow in a Spark Application and then how to configure the number of executors, memory settings of each executors and the number of cores for a Spark Job. There are a few factors that we need to consider to decide the optimum numbers for the above three, like:

* The amount of data
* The time in which a job has to complete
* Static or dynamic allocation of resources
* Upstream or downstream application

Introduction

Let’s start with some basic definitions of the terms used in handling Spark applications.

**Partitions** : A partition is a small chunk of a large distributed data set. Spark manages data using partitions that helps parallelize data processing with minimal data shuffle across the executors.

**Task** : 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. The unit of parallel execution is at the task level.All the tasks with-in a single stage can be executed in parallel

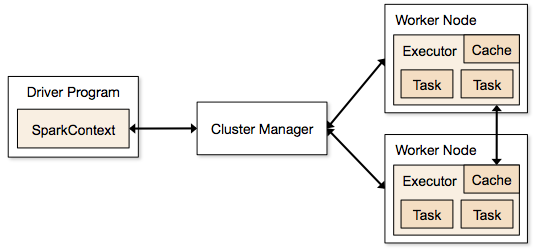
**Executor** : An executor is a single JVM process which is launched for an application on a worker node. Executor runs tasks and keeps data in memory or disk storage across them. Each application has its own executors. A single node can run multiple executors and executors for an application can span multiple worker nodes. An executor stays up for the  
duration of the Spark Application and runs the tasks in multiple threads. The number of executors for a spark application can be specified inside the SparkConf or via the flag –num-executors from command-line.

**Cluster Manager** : An external service for acquiring resources on the cluster (e.g. standalone manager, Mesos, YARN). Spark is agnostic to a cluster manager as long as it can acquire executor processes and those can communicate with each other.We are primarily interested in Yarn as the cluster manager. A spark cluster can run in either yarn cluster or yarn-client mode:

**yarn-client mode** – A driver runs on client process, Application Master is only used for requesting resources from YARN.

**yarn-cluster mode** – A driver runs inside application master process, client goes away once the application is initialized

**Cores** : A core is a basic computation unit of 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.



Steps involved in cluster mode for a Spark Job

1. From the driver code, SparkContext connects to cluster manager (standalone/Mesos/YARN).
2. Cluster Manager allocates resources across the other applications. Any cluster manager can be used as long as the executor processes are running and they communicate with each other.
3. Spark acquires executors on nodes in cluster. Here each application will get its own executor processes.
4. Application code (jar/python files/python egg files) is sent to executors
5. Tasks are sent by SparkContext to the executors.

From the above steps, it is clear that the number of executors and their memory setting play a major role in a spark job. Running executors with too much memory often results in excessive garbage collection delays

Now we try to understand, how to configure the best set of values to optimize a spark job.

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

1. Static Allocation – The values are given as part of spark-submit
2. Dynamic Allocation – 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.

Static Allocation

Different cases are discussed varying different parameters and arriving at different combinations as per user/data requirements.

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

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

***We start with how to choose 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 the ability of an executor to run parallel tasks and not from how many cores a system has. So the number 5 stays same even if we have double (32) cores in the CPU

***Number of executors:***

Coming to the next step, with 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 with 6 nodes, and 3 executors per node – we get a total of 18 executors. Out of 18 we need 1 executor (java process) for Application Master in YARN. So final number is 17 executors

This 17 is the number we give to spark using –num-executors while running from spark-submit shell command

***Memory for each executor:***

From above step, we have 3 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 full memory request to YARN for each executor.

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

Calculating that overhead:  .07 \* 21 (Here 21 is calculated as above 63/3) = 1.47

Since 1.47 GB > 384 MB, the overhead is 1.47

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 – 6 Nodes and Each node have 32 Cores, 64 GB

**Number of cores** of 5 is same for good concurrency as explained above.

**Number of executors** for each node = 32/5 ~ 6

So total executors = 6 \* 6 Nodes = 36. Then final number is 36 – 1(for AM) = 35

**Executor memory**:

6 executors for each node. 63/6 ~ 10. Overhead is .07 \* 10 = 700 MB. So rounding to 1GB as overhead, we get 10-1 = 9 GB

*Final numbers – Executors – 35, Cores 5, Executor Memory – 9 GB*

Case 3 – When more memory is not required for the executors

The above scenarios start with accepting 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 following are the numbers:

Cores: 5

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 10 is ok (assume little overhead), then we cannot switch the number of executors per node to 6 (like 63/10). Because with 6 executors per node and 5 cores it comes down to 30 cores per node, when we only have 16 cores. So we also need to change number of cores for each executor.

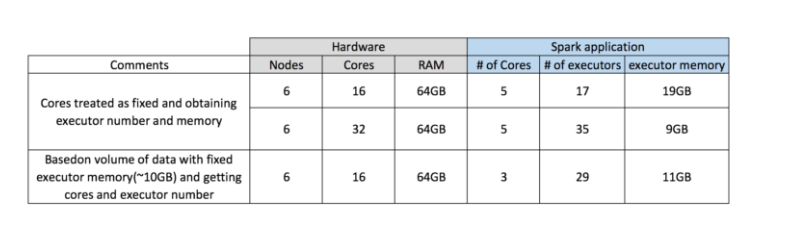
So calculating again,

The magic number 5 comes to 3 (any number less than or equal to 5). So with 3 cores, and 15 available cores – we get 5 executors per node, 29 executors ( which is  (5\*6 -1)) and memory is 63/5 ~ 12.

Overhead is 12\*.07=.84. So executor memory is 12 – 1 GB = 11 GB

*Final Numbers are 29 executors, 3 cores, executor memory is 11 GB*

Summary Table



Dynamic Allocation

*Note: Upper bound for the number of executors if dynamic allocation is enabled is infinity. So this says that spark application can eat away all the resources if needed. In a cluster where we have other applications running and they also need cores to run the tasks, we need to make sure that we assign the cores at cluster level.*

*This means that we can allocate specific number of cores for YARN based applications based on user access. So we can create a spark\_user and then give cores (min/max) for that user. These limits are for sharing between spark and other applications which run on YARN.*

To understand dynamic allocation, we need to have knowledge of the following properties:

*spark.dynamicAllocation.enabled –*when this is set to true we need not mention executors. The reason is below:

The static parameter numbers we give at spark-submit is for the entire job duration. However if dynamic allocation comes into picture, there would be different stages like the following:

**What is the number for executors to start with:**

Initial number of executors (*spark.dynamicAllocation.initialExecutors*) to start with

**Controlling the number of executors dynamically:**

Then based on load (tasks pending) how many executors to request. This would eventually be the number what we give at spark-submit in static way. So once the initial executor numbers are set, we go to min (*spark.dynamicAllocation.minExecutors*) and max (*spark.dynamicAllocation.maxExecutors*) numbers.

**When to ask new executors or give away current executors:**

When do we request new executors (*spark.dynamicAllocation.schedulerBacklogTimeout*) – This means that there have been pending tasks for this much duration. So the request for the number of executors requested in each round increases exponentially from the previous round. For instance, an application will add 1 executor in the first round, and then 2, 4, 8 and so on executors in the subsequent rounds. At a specific point, the above property max comes into picture.

When do we give away an executor is set using *spark.dynamicAllocation.executorIdleTimeout.*

To conclude, if we need more control over the job execution time, monitor the job for unexpected data volume the static numbers would help. By moving to dynamic, the resources would be used at the background and the jobs involving unexpected volumes might affect other applications.