# **How We Optimise Apache Spark Jobs**

Here in Consumer Insights we have been operating Big Data processing jobs using [Apache Spark](https://spark.apache.org/) for more than 2 years. Spark empowers our daily batch jobs which extract insights from consumer behaviors from tens of millions of users who visit our sites. This blog covers our usage of Spark and aims to provide some useful insights for optimizing Spark applications based on our experience.

Spark is designed to run large scale data processing applications on clusters of machines, in which it distributes the workload to achieve much faster run time. Despite the fact that Spark is "lightning-fast" due to its in-memory processing and is generally more performant than the other cluster computing frameworks—like Hadoop MapReduce—we had faced issues in the past with some of our Spark jobs often failing, getting stuck, and taking long hours to finish.

Given the number of factors and parameters that can impact Spark’s performance, optimisation has been a continuous journey for us. Early this year, we had opportunities to re-visit some of our data processing jobs to make them run faster, more stable, and more cost-efficient.

Spark on EMR and YARN

We deploy Spark jobs on [AWS EMR](https://aws.amazon.com/emr/) clusters. An EMR cluster usually consists of *1*master node, *X* number of core nodes and *Y* number of task nodes (*X* & *Y* depends on how many resources the application requires) and all of our applications are deployed on EMR using Spark's cluster mode. Here we have another set of terminology when we refer to containers inside a Spark cluster: Spark driver and executors. In cluster mode, Spark driver is run in a YARN container inside a worker node (i.e. one of core or task EMR nodes). When deployed in client mode, Spark driver is run inside the master node of EMR (i.e. outside of the cluster). In this blog, examples are demonstrated in the context of cluster mode.

A simplified view of Spark cluster mode on EMR

Cluster resources (CPUs and memory) are managed by [YARN](https://hadoop.apache.org/docs/r2.7.2/hadoop-yarn/hadoop-yarn-site/YARN.html) through pre-installed Hadoop environment in EMR. YARN is responsible for managing and allocating resources to separate containers for distributed workload. To simplify, each YARN container has a number of virtual cores (vCores) and allocated memory. A JVM will be launched in each of these containers to run Spark application code (e.g map/reduce tasks). Therefore, multiple Spark tasks can be run concurrently in each executor and available executors can run concurrent tasks across the entire cluster.

Spark is great, but it also comes with extra complexity to be dealt with, namely EMR and YARN configuration, cluster sizing, memory tuning, algorithms, data structure, data serialisation, and garbage collection, amongst others. But we have found that performance can be boosted by addressing some of the common problems described in the following sections.

Before we deep dive into addressing potential issues, the first thing we need to know is whether a Spark job is running in a healthy and efficient state.

Monitoring Cluster Metrics

In this section, I will briefly cover tools that help diagnose problems in Spark jobs.

Spark cluster runtime status can be monitored via:

* [Spark Web UI](https://spark.apache.org/docs/latest/monitoring.html#web-interfaces), which provides a view of all scheduled tasks and spark configurations. Symptoms of an unhealthy state can be: a single task getting stuck for an extensive period of time, or a single task failing due to Spark exceptions.
* [Ganglia](http://ganglia.info/), which provides a view of cluster resources usage, including memory and CPU. Symptoms of an unhealthy state can be: low percentage of CPU usage, a large number of idle CPUs, or memory spikes.
* YARN ResourceManager UI, which provides a view of cluster resources, including the number of executors, memory and CPUs per executor. Symptoms of an inefficient state can be: the actual number of executors lower than expected, or the allocated memory/ CPUs lower than expected.

They are all built-in tools which come with installing Spark, Hadoop and Ganglia on EMR (see [View Web Interfaces Hosted on Amazon EMR Clusters - Amazon EMR](http://docs.aws.amazon.com/emr/latest/ManagementGuide/emr-web-interfaces.html)).

Now let's look at how we fixed some of the common problems.

Underutilised Cluster

Based on how Spark works, one simple rule for optimisation is to try utilising every single resource (memory or CPU) in the cluster and having all CPUs busy running tasks in parallel at all times. The level of parallelism, memory and CPU requirements can be adjusted via a set of Spark [parameters](https://spark.apache.org/docs/latest/configuration.html), however, it might not always be as trivial to work out the perfect combination.

In the past, there were two approaches to setting parameters in our Spark job codebases: via EMR's [maximizeResourceAllocation](http://docs.aws.amazon.com/emr/latest/ReleaseGuide/emr-spark-configure.html" \l "w2ab1c46c81b9) and manual configuration. spark.dynamicAllocation.enabled could be another option but we don't really need it as we don't run multiple applications on a shared cluster. Based on our experience, we have found that using maximizeResourceAllocation is not optimal and we prefer to configure those settings manually.

Approach A: maximizeResourceAllocation

According to its documentation, maximizeResourceAllocation will try to auto-configure the following:

spark.executor.cores = number **of** CPUs on a worker node

spark.executor.instances = number **of** worker nodes on a cluster

spark.executor.memory = max memory available on a worker node - overheads

spark.default.parallelism = 2 \* number **of** CPUs **in** total on worker nodes

However, it does not seem to always work out the ideal numbers when we use some instance types (e.g. *R4* instances). Here is an example of our job which ran on 10 *r4.4xlarge* core nodes with a total of 160 [vCPUs](https://aws.amazon.com/ec2/virtualcores/)(or hyperthreads) and 1.11TB memory (runtime cluster metrics are shown below).

a snapshot of YARN ResourceManager UI showing a big proportion of unused resources

At first glance, YARN launched only 6 containers (5 executors + 1 driver) out of 10 nodes and each container was assigned with 8 cores (vCPUs) and around 10GB. Further investigation showed that internally spark.executor.memory was set to 9658M (plus 10% memory overhead = 10.37 GB) and spark.executor.cores was set to 8. This is wasteful in that the majority of memory and more than half of CPUs were not used. In addition, it kept throwing OutOfMemory errors, which resulted in shutting down the cluster. The auto-configured resources were too limited for a job designed to process over 300GB of raw data and involved quite a lot of compute-intensive operations.

Approach 2: manual configuration

Based on our experience, most of the time we would be able to maximise cluster utilisation by getting these few key parameters right:

spark.executor.memory

spark.executor.cores

spark.executor.instances

spark.yarn.executor.memoryOverhead

spark.default.parallelism

In order to calculate all these specifics, let's imagine we have a cluster of 8 *r4.2xlarge*nodes, each node with 8 vCPUs and 61 GB memory.

spark.executor.cores tells spark how many concurrent tasks that can be run in each executor, and that usually means the number of CPUs to be assigned for each executor. One approach (see diagram illustrated below) would be assigning as many CPUs (spark.executor.cores = 8 in our example) as possible to each executor. However, as a result, there will be only 7 executor instances that can be launched, each inside a *r4.2xlarge* node and spark driver will take up 1 core in the last r4 instance.

Manual configuration version 1

This is not ideal because cluster resources i.e. CPUs are not fully utilised and [experience](http://c2fo.io/c2fo/spark/aws/emr/2016/07/06/apache-spark-config-cheatsheet/) shows that executors running with more than 5 concurrent tasks, may lead to poor HDFS I/O throughput.

A better approach (see diagram *Manual Configuration 2* below) would be having multiple smaller executors launched inside a node, let's say spark.executor.cores = 2. To illustrate, we will have maximum 4 executors in each *r4.2xlarge* instance and 3 executors in the instance where spark driver will be run. In order to make sure that YARN will launch as many executor instances as possible, we can set spark.executor.instances = 7 \* 4 + 3 = 31. Be aware that sometimes having many tiny executors also has some drawbacks as it will end up having more JVMs running (overheads) and more copies of data if [broadcast variables](https://spark.apache.org/docs/latest/programming-guide.html#broadcast-variables) are used.

Manual configuration version 2

spark.executor.memory specifies the amount of memory to use per executor. Because Spark processes data in memory, the more memory an executor has, the less frequent/likely data may spill to disk which incurs slow disk I/O. So, the rule for us is always the more the better, but we tend not to use 100% memory on a node for executors as it needs resources for running OS and Hadoop daemons.

By default, EMR tells YARN the maximum memory that can be allocated on a single node based on each instance type (see [Task Configuration - Amazon EMR](http://docs.aws.amazon.com/emr/latest/ReleaseGuide/emr-hadoop-task-config.html#emr-hadoop-task-jvm)). When calculating executor memory, it also needs to take spark.yarn.executor.memoryOverhead into account. By default,

spark.yarn.executor.memoryOverhead = spark.executor.memory \* 0.10.

Putting them together, spark.executor.memory can be calculated by:

spark.executor.memory + spark.yarn.executor.memoryOverhead = memory per node / number **of** executors per node

Hence,

spark.executor.memory = 54272 \* / 4 / 1.10 ~= 12335M

The last step is to determine spark.default.parallelism which controls the number of data partitions to be generated after certain operations (e.g. join, groupBy, aggregateBy, etc) in order to reduce the size of shuffle data structures. More importantly, it impacts the number of tasks to be run in parallel.

For example, if the number of data partitions is less than the total number of cores available, Spark cannot take full advantage of utilising cluster resources. In general, we usually set parallelism to be at least 2~4 times of spark.executor.instances \* spark.executor.cores, so that there will be enough concurrent tasks to keep executors busy. In some other cases, some of the tasks may incur heavy disk I/O due to aggregating large data structures and slowed down the job. This can be optimised by increasing parallelism by a larger multiplier. Just keep in mind that having way too many partitions will also sacrifice its performance because there are small overheads for Spark managing many small tasks.

spark.executor.memory = 12335M

spark.executor.cores = 2

spark.executor.instances = 31

spark.yarn.executor.memoryOverhead = 1396

spark.**default**.parallelism = 62

Unnecessary Data Shuffles

As you may already know, data shuffles are expensive. Common shuffle operations like join, groupBy, aggregateBy will move data across the network and put them in the same partitions based on keys. Because groupByKey only groups data after data shuffles hence it usually transfers bigger trunks of data over the network and slows down the performance (examples demonstrated in this [Databrick article](https://databricks.gitbooks.io/databricks-spark-knowledge-base/content/best_practices/prefer_reducebykey_over_groupbykey.html)). Therefore we eliminated usage of groupByKey and instead we prefer to use aggregateByKey / reduceByKey which combine keys in each partition before shuffle data globally.

Out Of Memory Error

Sometimes, we had multiple incidents of Spark crashes due to OutOfMemoryError. This is probably one of the most common causes of Spark failure among Spark users. Digging through logs you will probably find something like this:

It is mentioned in the previous section that spark.yarn.executor.memoryOverhead is set to be 10% of executor memory by default. Such errors suggest that an application has used up all off-heap memory for overheads. While it can be considered as a hint that the Spark application code is not written efficiently, one possible way to fix this issue without refactoring the code is to gradually increase memory overhead (e.g. 15% -> 20% -> 25%) and re-calculate other specifics.

Use Bigger Instance Type (Wisely)

If your Spark job does a lot of heavy data crunching and causes frequent data spills to disk, you probably will have to run it on a bigger cluster. This does look obvious—by upgrading instance type you will get more CPUs and memory, then you can increase the number of executors and cores!

But, you might wonder, what about the cost? EMR can be costly because applications are run on clusters of machines for a fair number of hours. Using spot instances is a primary way of cost saving.

Saving on spot instances varies depending on instance types and the demands. For some popular instance types (e.g. *r3.2xlarge*, *m3.2xlarge*), we usually get around 70% savings but sometimes we would be at risk of constantly getting outbid due to high demand.

We have found that applications can run on other EC2 instances with similar or even better specs for a much lower price. For example, the next generation R4 instances have the exact same specs as R3 instances (except for EBS-only storage) and their spot instances provided up to 85% savings.

Spot instance price

From the spot price snapshots above, the price of *r4.4xlarge* (16 CPUs, 122GB memory) was almost the same as *m3.2xlarge*'s (8 CPUs, 30GB memory) and just a bit more than *r4.2xlarge* (8 CPUs, 61GB memory). Compared with R3 instances, the price of *r4.4xlarge* was 34% lower than *r3.2xlarge*'s.

Another good strategy is to test the Spark job on multiple instance types during development and define the instance capacity pool (e.g. Application A can run on *r3.2xlarge*, *r3.4xlarge* or *r4.2xlarge*). Then it can pick and choose to run on the cheapest instance type based on the live spot price. This also reduces the risk of EMR interruptions or instances unavailability due to price spikes.

The Results

By applying a mix of optimisations mentioned above, we managed to further reduce runtime and cost for most of our Spark jobs.

In details (see *Performance before vs after*), we cut-down the runtime for our current biggest Spark job *Buy-rec*, which runs on a 21-node cluster) by around 1 hour (around 20% time reduction). It is worth noting that we adapted both *Buy-rec* and *Rent-rec* to process 40% more data than their pre-optimisations. The optimisation also fixed a number of unstable applications (*IIPs*, which is a collection of 14 Spark applications used for feature extraction), which used to randomly fail and get stuck for hours, and improved their run-time by 50%.

Performance before vs after

Final thoughts

Performance tuning requires extra time and efforts to understand what input data the job deals with, how Spark works with different cluster environments, and what the application code tries to achieve. Although this blog focuses mainly on the configuration side of optimisation, I do think that optimisation can also be done by writing efficient Spark code and using efficient data structures