5 Key Factors to keep in mind while Optimizing Apache Spark in AWS(Part 1):

This article aims to help experienced developers with some of the bottlenecks faced while dealing with extreme volume of data with limited resources. It is not about fundamentals and theoretical optimization techniques which are frequently discussed. Suggested solutions( or optimization tricks) are based on inferences drawn from the practical problems faced while optimizing Apache Spark.

**Long Lineage**

Lazy evaluation in spark means, actual execution does not happen until an action is triggered. The types of commands available in spark can be divided into 2 types.

* Actions ( eg. *head(), show(), write(), count()*)
* Transformations (eg. *map(), filter(), groupBy(), select()*)

Every transformation command run on spark gets added to the lineage(*explained below*) after the syntax check, actual execution happens only when an action based command is run.

***Optimization Trick***: It is not advisable to chain lot of transformation in a lineage, especially when you would like to process huge volume of data with minimum resources. Rather, break the lineage by writing intermediate results into HDFS( preferable HDFS if you have storage available, as writing S3 could be slower)

**File System Preferences**

The types of files we deal with can be divided into two types

* Splittable ( eg. LZO, Bzip2)
* Non- Splittable ( eg. Gzip, Zip)

For the purpose of the discussion, Splittable files means they are parallely processsable in a distributed fashion rather in one machine( non-Splittable).

***Optimization Trick***: If you have a huge file (10gb and zipped) and you try to load into spark, it might just get processed using one node( or executor) if it is not splittable which could be a bottleneck. If you come across such cases, it is a good idea to use s3cmd and move the file from s3 into HDFS and unzip it(If the big file you are referring is in s3). If it is in HDFS, you could unzip it before you load into spark.

Note : We will discuss the columnar file formats in PPD section below.

**Writing Queries and/or Transformations**

The biggest mistake people make in big data systems is, try to “optimize queries” in fact it should be “optimize data”. “Simplicity is the Key”, This is applied to all distributed systems including spark. To apply this in real life, it is advised not to write complex queries in spark, rather try to break it down as much simpler steps as you can. People have a misunderstanding that, more number of steps could increase the processing but, actually not. Spark might internally combine some of the steps and perform at once.

***Optimization Trick***: Always try to break your queries (or transformations) into granular steps instead of writing one big query. Operations chained in spark are different steps ( not a single big query or transformation)

**Predicate Push Down(PPD)**

PPD in simple terms, is a process of only selecting the required data for processing when querying a huge table. eg: If you have a table of 100 columns and you are querying only 10 columns, in PPD data for only those 10 columns are selected for further processing. Another example could be, if there is a filter clause(eg. where clause) in any query, the filter will be applied first to reduce the number of records picked for processing. This significantly improves the performance by reducing the number of records read/write resulting reduction in input/output operation.

Columnar file formats give us a great way of using the power of PPD as it inherently enabled to do so. Some of the examples of Columnar file formats are Parquet, RC or Row-Column, ORC or Optimized Row-Column etc.

***Optimization Trick***:There are two important notes to make here.

* Use Parquet format wherever feasible for reading and writing files into HDFS or s3 as parquet seems to be performing very well along with Spark. Especially, All the intermediate steps that you would like to write data into HDFS so as to break the lineage( As mentioned under optimization trick in Lazy Evaluation)
* Always try to identify the “filters” and try to move it up as early as you can for all your data processing pipeline.

**Data Skew Checks**

Performance of the distributed systems are highly dependent on how much distributed the data is. One way to ensure distribution is to check the number of partitions of a RDD or a DataFrame.

***Optimization Trick***: Do check the number of paritions of the dataframes or RDDs just before you carry out any complex operation. In case you find the number of partitions are too low, it is a good idea to repartition them to increase the number of partitions. you could use the below line of code for checking the number of partitions in pyspark.

df.rdd.getNumPartitions()

**Conclusions**

In Bigdata systems it is advisable to optimize data first before we think about optimizing quries.

The second part of the story is available on the below link. Kindly, give a read and share your feedback.

<https://medium.com/@brajendragouda/5-key-factors-to-keep-in-mind-while-optimising-apache-spark-in-aws-part-2-c0197276623c>

#### Join Operations

During joining if you have a big table and a relatively small table ( lookup or dimension table) it is advisable to broadcast the small table. In broadcasting a copy of the broadcasted table is sent to each node of the cluster. So, while joining, part of the bigger table there in a node joins with the broadcasted table therefore does not move data across nodes and reduces I/O operations hence improves performance.

**Optimisation Trick:**

if you are joining a Big table with a small one then it is good idea to broadcast the smaller table.But keep in mind that, the smaller table should be small enough to fit inside memory of an executor. If both the table you are trying to join are big and similar in size, then ensure both the tables are not skewed or distributed across more number of partitions. If not, repartition to increase number of partitions of the skewed table.Out of the two tables if you find that one of tables is not similar in size to the other table and not small enough to be broadcasted, then you could cache( or persist) the smaller table and ensure bigger table is partitioned properly before performing Joins.

#### Maximising Parallelism

One way to increase parallelism of spark processing is to increase the number of executors on the cluster. Below are 2 important properties that controls number of executors.

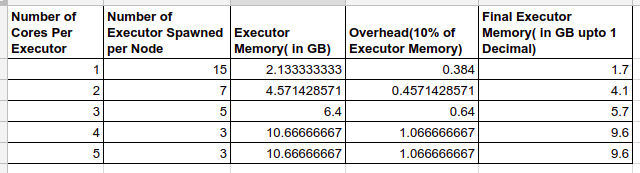
spark.executor.memory # Amount of memory to use per executor process  
spark.executor.cores # number of cores to use on each executor

Let us take an example to understand how these two properties are used to decide how many executors should be spawned.

Consider a Hardware with 5 Nodes, each with 16 Cores and 32 GB. Before we calculate number of executors, few things to keep in mind.

* A Node can have multiple executors but not the other way around.
* An Executor can have multiple cores.
* property spark.executor.core should only be given integer values.
* property spark.executor.memory can have integer or decimal values upto 1 decimal place.
* Not advisable to have more than 5 cores for each executor. This is based on a study where any application with more than 5 concurrent threads would start hampering the performance.

Some resources are needed for OS and Hadoop Daemons, say around 1 core and 1 GB Memory needs to be allocated. we are left with 15 Cores and 31 GB. Since we can not consider fraction of cores for executors, at max we can 15 executors i.e 1 for core for each executor. Each executor would also need some memory for overhead such as VM overhead, interned string etc while communicating with the Master (Yarn in case of AWS). This is usually 10% of the executor memory with minimum of 384 MB. The table below will give us some idea about how the number of executors could vary based on different parameters.



From the above table, if we have 1 core per executor we can have 15 executors, each with 1.7GB of memory. More the number of executors means better parallelism and more memory per executor means a bigger chunk of data can be processed in each executor.

**Optimisation Trick:**

A balance has to be maintained between number of cores for each executor to that of executor memory. Although, understanding of data and complexity of the algorithms is a driving force to identify the balance, most of the times selecting something from the middle the table would be optimal. from the above table 5 executors (in row 3) 3 cores and 5.7 GB per executors will be a good idea. If our algorithms are complex and iterative ( in most of Machine Learning algorithms) it is good to select something end of the table ( either 4 cores or 5 cores per executor) on the other hand if volume of data is very high but algorithms are not that complex and iterative than selecting something middle of the table (3 cores) should perform better.

#### User Defined Functions

This is for people who are from R/Python background. They write functions which accepts/returns dataframes. Although this syntactically works and returns results, it downgrades the performance of the system. Let us take an example and understand how not to write UDF in spark.

Not recommended way of writing UDF in PySpark

## A function that accepts a dataframe and converts a specific column of a dataframe into upper case.

def myfunc(df, sub\_str):  
 df['col\_2'] = df['col\_1'].apply(lambda x: upper(x), axis=1)  
 return df

Correct way of writing UDF in PySpark

## A function that accepts a dataframe and a substring as a parameter and does some string operation

def myfunc(str):  
 return upper(str)

myfuncUdf = udf(myfunc, StringType())  
# Call the function in spark dataframe  
df = df.withColumn('col\_2', myfunc(df.col\_1))

Few things to note from the above two ways of writing UDF in spark.

* First one is python way and second one is Spark(or PySpark) way
* Python way, we have a dataframe as an argument. In Spark way, function takes one record as an argument.
* In Spark way, the function works in a distributed way and parallely executes in all executors.

**Optimisation Trick**

While writing UDF assume the function accepts one row returns one row. we can input multiple columns but one row only. If you have more than two arguments(columns) to your UDF , I would advice to create one array using all your arguments and pass it to the UDF.

#### Monitoring Cluster Metrics

Amazon EMR provides built in tools for monitoring cluster metrics which can be selected for installation while starting a cluster.

* Spark Web UI- It is available by default when we select Spark and Hadoop in EMR software configurations. [Use this](https://docs.aws.amazon.com/emr/latest/ManagementGuide/emr-connect-ui-console.html) for your reference. Using Spark UI we can view all scheduled tasks and configurations.
* Ganglia- It can be selected during cluster creation in table Create Cluster ->Advanced -> Software Configuration step. This is useful in understanding cluster resource usage like CPU, Memory etc.
* Yarn Resource Manager UI- Yarn is the default Master for Spark in EMR. Yarn UI gives lot of information about cluster resources including number of executors, CPU and memory per executor.

**Optimisation Trick**

Monitor cluster metrics using any of the above tools and proactively fix any performance issues. Symptoms like a single task getting stuck for significant amount of time or a task failing due to spark exceptions are clear indication of unhealthy states which can be identified using Spark UI. Low percentage of CPU usage, larger number of idle CPUs or memory spikes are symptoms of unhealthy state identified through Ganglia. Actual number of executors lower than expected, or allocated memory/CPU lower than expected are indication of unhealthy states identified through Yarn Resource Manager UI.

#### Explain Plan

Another way of identifying potential bottlenecks in spark is by using Explain query plan.

df.explain()  
df.explain(True)

explain() prints the physical plan whereas explain(True) prints Logical, Analysed, Optimised and Physical plan of the query. The logical plan is a tree that represents schema and data, which is of three types

* Parsed logical plan
* Analysed logical plan
* Optimised logical plan

Optimised logical plan converted into physical plan for execution.

**Optimisation Trick**

Look into all the plans above and identify opportunities for optimisation. Avoid full table scans if possible, apply filters as early as you can in the processing steps and ensure lineage isn’t long before performing joins.

#### Conclusion

Although there are lot of inbuilt optimisations already available in spark, it is necessary to smartly use all of them to get best out of it.

Here is first part of the story, please give a read and let me know your thoughts.