Lecture Notes

Optimising Spark for Large-Scale Data Processing

In this module, you learnt how to optimise Spark in terms of both execution time and resource utilisation. You first learnt about Amazon EMR and how this service can be used to set up a multi-node Spark cluster. Then you learnt about the anatomy of a Spark job and understood why there is a need for optimising Spark jobs in the industry. You learnt about the various approaches that can be adopted to optimise Spark jobs. You started with learning how to optimise Disk IO in Spark jobs using various techniques such as file format optimisation, use of optimised Serializers, and managing Spark Memory management parameters, including Persist, Cache and Unpersist. Then in the subsequent session, you learnt about the various methods for optimising Network IO, such as using reduceByKey instead of groupByKey, optimising joins, using optimised partitioning techniques and using Custom partitioners. Finally, after learning how to optimise Spark jobs at the code level, in the last session, you learnt how to optimise a Spark cluster itself. Here, you first learnt why it is important to optimise the Spark cluster configuration and how to avoid underutilisation of cluster resources. You learnt about the various job deployment modes in Spark and also learnt about the various parameters that you can optimise for a Spark cluster. You also learnt how to maintain an appropriate cost–performance trade-off. Finally, you learnt how Spark jobs are deployed in the industry and also learnt about some of the best practises that you should follow while working with Spark.

Spinning Up a Spark EMR Cluster

So far, you have run Spark jobs on the CDH EC2 instance. The CDH EC2 instance is a single-node cluster, and it is quite expensive if you try to maintain a multi-node cluster for long periods of time. This is where **Amazon EMR (Elastic MapReduce) clusters** come into the picture.

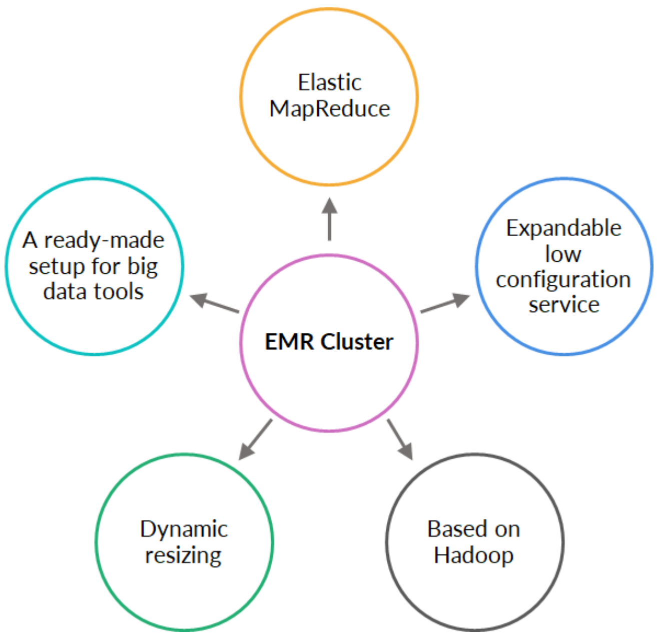

Elastic MapReduce
A ready-made setup for big data tools
Expandable low configuration service
EMR Cluster
Dynamic resizing
Based on Hadoop

Figure 01: Features of an EMR Cluster

Amazon EMR is a managed cluster platform, which offers an expandable low-configuration service. It also provides ready-made big data tools depending on the cluster configuration that you choose.

Amazon EMR supports **Dynamic Resizing**, which means it has the ability to scale the number of nodes automatically on the fly. You can also control such parameters manually when running Spark jobs. EMR clusters come with the following features, which are also their major advantages over EC2 instances:

- They provide automatic scaling and ensure minimal loss of HDFS data. Also, since generally spot instances are used with EMR, they are generally cheaper than EC2 instances of similar configuration over long periods of time.
- They allow dynamic orchestration of new clusters on demand, along with easy termination of clusters once the work is complete. The actual steps involved in the creation of an EMR cluster are quite straightforward, and most of the configuration takes place in the background. You can also use the features of **AWS Step Functions** to automate the whole process of building the entire data processing and analysis workflows with minimal code.
- They enable direct access to data on S3 and data on other connected AWS services, such as Hive tables.
- They ensure high availability of slave nodes by constantly monitoring each node and automatically replacing all unhealthy nodes with new nodes.

However, it is important to note that EMR clusters are not the perfect alternative to Amazon EC2 machines and have some drawbacks as well. Some of these drawbacks are as follows:

- EMR does not have a management console similar to that of Cloudera Manager; this makes it much harder to manage and monitor the various services.
- Even though EMR ensures high availability of slave nodes, it does not ensure the same for the cluster's master node; this makes it a single point of failure.
- One of the biggest disadvantages of using EMR clusters is that you cannot shut them down; you need to terminate them directly. Since data stored on EMR clusters is lost upon termination, you need to create a backup for the data that you want to save in S3 buckets. Fortunately, EMR clusters automatically save the Jupyter Notebooks that you create in S3 buckets.
- Although automatic replacement of unhealthy nodes ensures high availability of slave nodes, it creates a high possibility of loss of the data that is present in the unhealthy nodes.

Analysing Spark Jobs

While working in Spark, you will come across mainly four different terms. These are as follows:

- **Application**: An application is the highest level of computation in Spark. It is the main function that contains all the code and can be run by the user to compute some results.
- **Job**: Whenever an action is called in the application, the work that is carried out is called a Spark job. It is important to note that an application may consist of multiple jobs.
- **Stages**: Whenever a shuffle operation takes place inside a Spark job, it creates a new Stage.

- **Tasks**: Tasks are the most basic unit of work in Spark computation. Each stage has multiple tasks.

Spark provides a utility called the Spark History Server UI where you can easily check the various tasks and stages, and how they are executed for your Spark job.

Why Optimise Spark Jobs?

The need to optimise Spark jobs can be viewed from different perspectives:

- **Execution time:** At the industry level, optimising Spark jobs can reduce the execution time and, thereby, increase productivity. Generally, execution time affects the performance of Spark jobs; hence, this can prove to be of great advantage.
- **Resource utilisation:** Since we do not have access to infinite resources, we have to ensure that the resources available are being utilised to their maximum potential in order to maximise efficiency.
- **Scalability:** Data grows at an exponential rate in the industry, in terms of both volume and velocity. So, it is important that Spark jobs are optimised in such a way that they are able to handle the increasing demand.
- **Maintainability:** Typically, in the big data industry, most Spark jobs are not run just once and then set aside. Instead, they are supposed to be scheduled and reused on a periodic basis. Therefore, it is important to optimise Spark jobs to ensure maintainability, so that they can be reused easily in the data pipeline even if there are some errors.

Some of the key performance metrics that should be considered while optimising Spark include the following:

- **Stages and tasks:** Since these are the basic units of work in a Spark job, it is important to closely monitor their performance. A higher number of stages means more shuffling, and this adversely affects the Network IO and, in turn, the overall performance of a Spark job.
- **Resilient Distributed Dataset (RDD) (memory imprint and usage):** By monitoring the usage of memory and CPU utilisation, you can prevent OutOfMemoryError in your Spark jobs.
- **Spark environment information:** Some of the metrics of the Spark environment, such as the different configurations and the root locations of libraries, are useful for analysing the performance of Spark jobs.
- **Detailed information about running executors:** Executors are the actual nodes on which Spark jobs are run. So, metrics such as the memory imprint of the nodes, CPU utilisation and the number of Executors running can help in understanding important factors, including the degree of parallelism for Spark jobs.

When it comes to optimising an actual job, we mainly perform the following two different types of optimisation:

- **Code-level optimisation**: This includes techniques such as deciding how many partitions to create and what the partitioning size would be, which APIs to use for handling data and which methods to use.
- **Cluster-level optimisation**: This includes techniques such as deciding the optimal number of machines, improving the utilisation of a cluster and preventing underutilisation.

In code-level optimisation, you can optimise a job in the following two different ways:

- **Reducing Disk IO:** Disk IO is one of the biggest challenges that are faced while optimising the performance of a Spark job. It is also one of the slowest processes in a Spark job. The performance of a Spark job can be improved significantly by avoiding unnecessary Disk IO.
- **Reducing Network IO:** Network IO is another process that becomes quite slow whenever shuffling is being carried out. This is because data gets transferred between different nodes. Reducing network IO significantly improves the performance of Spark jobs.

In cluster-level optimisation, you can optimise parameters such as Executor memory and cores; this will directly improve the performance of Spark jobs. You need to optimise these parameters in such a way that they are neither too low, which may cause OutOfMemoryErrors, nor too high, which will lead to wastage and underutilisation of resources.

Understanding Disk IO in Spark

Disk IO is the process of fetching data or writing data to secondary storage devices such as hard drives (in commodity hardware, these hard disks are of magnetic type).

The entire process of Disk IO is much slower than other processes, owing to the nature of hard drives.

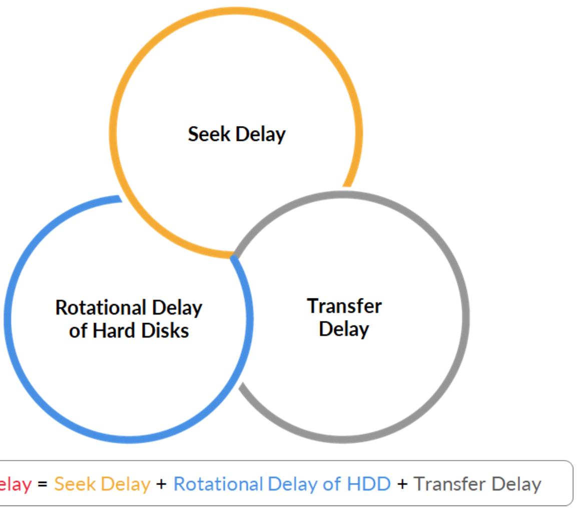

Seek Delay
Rotational Delay of Hard Disks
Transfer Delay
Total Delay = Seek Delay + Rotational Delay of HDD + Transfer Delay

Figure 02: Delays in Disk IO

The total delay caused by Disk IO is higher than the delays caused by other processes because hard drives are much slower than primary memory, such as RAM.

Reducing possible Disk IO can be quite beneficial in increasing the performance of Spark jobs.

There are several techniques to reduce Disk IO. Some of them are as follows:
- **Avoid shuffling as much as possible:** The shuffling process leads to the creation of stages. When this happens, the data at a stage boundary is stored in the disk so that it can be fault-tolerant.
- **Using optimised file formats (Parquet and ORC):** Use of optimised file formats, such as Parquet and ORC, can not only help reduce the size of files, but also help with the process of reading data. This is because these file formats are columnar in nature.
- **Serialization and deserialization:** Use of appropriate serialization and deserialization techniques for memory storage and cached data helps in reducing Disk IO.

Using Various File Formats in Spark

There are many file formats that you can use while writing Spark jobs. Some of the common file formats that are used in Spark include **.txt, .csv, .json, .avro, Parquet** and **ORC**.

The **Text** file format is available in just about every technology device that we use today, and it is the most common file format. **Comma-separated values** (CSV) files are another common file format that is used for storing data sets. **Javascript Object Notation (JSON)** has a dictionary-like structure in which data is stored in a key–value format. In an **Avro** file, data is stored in a particular schema.

Both Parquet and Columnar file formats are columnar in nature.

Here are some important points about the Optimised Row Columnar (ORC) file format:
- A typical ORC file is divided into two different segments: Index Data and Row Data; and Stripe Footer.
- It is generally used for both compressed and uncompressed storage.
- It stores collections of rows in a file, and within a collection, row data is stored in a columnar format.
- An advantage of using this file format is that the response time in both Reads and Writes is quite fast.
- This file format is generally preferred when the original data is flat and non-hierarchical.
- Files stored in this format may be compressed by as much as 70% in terms of file size.
- It also supports the lightweight index, which helps in improving the read time of data.

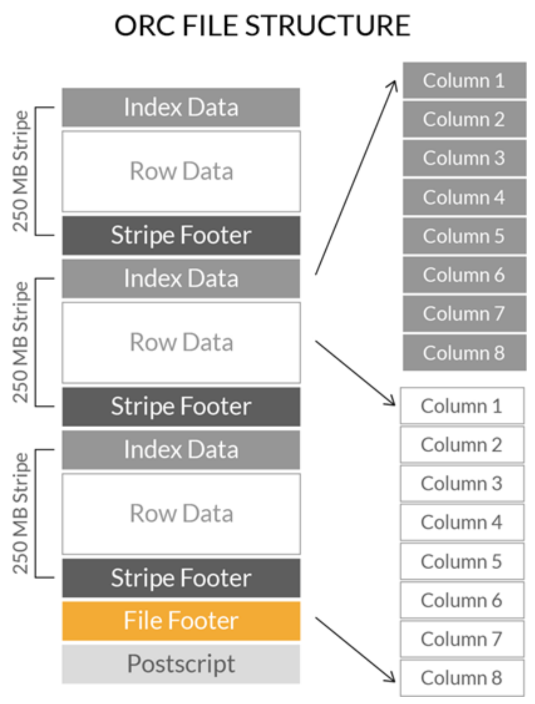


Figure 03: ORC File Structure

Here are some important points about the Parquet file format:

- Files stored in this format may be compressed by as much as 70% in terms of file size.
- The metadata of a file written in this format is attached at the end of the file.
- It is widely supported by all Apache big-data-processing tools.
- This file format is preferred when data is in a nested format.

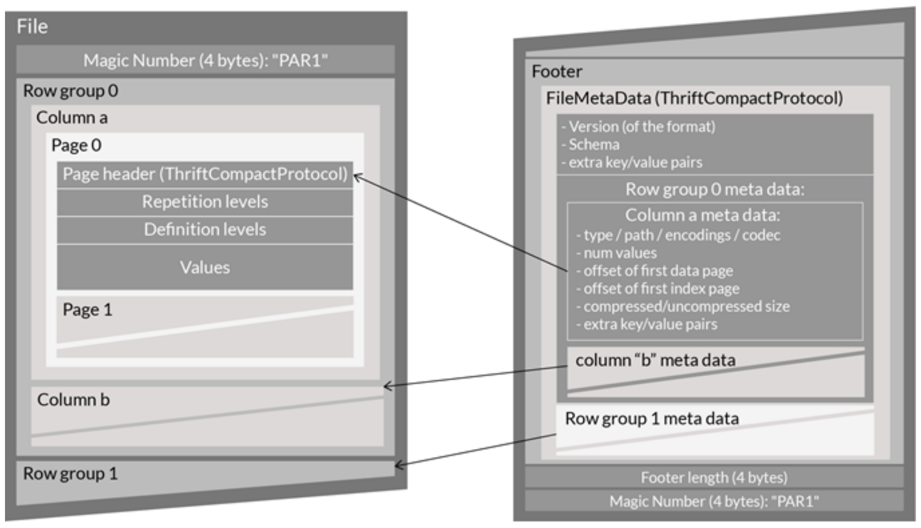

File
Magic Number (4 bytes): "PAR1"
Row group 0
Column a
Page 0
Page header (ThriftCompactProtocol)
Repetition levels
Definition levels
Values
Page 1
Column b
Row group 1
Footer
FileMetaData (ThriftCompactProtocol)
- Version (of the format)
- Schema
- extra key/value pairs
Row group 0 meta data:
Column a meta data:
- type / path / encodings / codec
- num values
- offset of first data page
- offset of first index page
- compressed/uncompressed size
- extra key/value pairs
column "b" meta data
Row group 1 meta data
Footer length (4 bytes)
Magic Number (4 bytes): "PAR1"

Figure 04: Parquet File Structure

The choice of a particular file format has a significant effect on the performance of your Spark job. Following are some of the ways in which file format affects the performance of a Spark:

- **Faster read time:** At the industry level, it is extremely important to read files as fast as possible; therefore, the performance of a file format is often judged based on its read time.
- **Faster write time:** Just like read time, write time is another straightforward parameter based on

which a particular file format is chosen in the industry. A faster write time has a major impact on the overall performance of a Spark job.

- **Splittable files:** Certain file formats support the feature of splitting a file into multiple smaller chunks. This has a major impact on the performance of jobs in the industry as it directly increases the degree of parallelism of your Spark job.
- **Schema evolution support:** In many industry use cases, you might need to accommodate certain changes in the schema of the data. There are various file formats that support this form of schema evolution.
- **Advanced compression support:** In certain situations, you might need advanced compression support for your Sqoop jobs. In such cases, you can use columnar file formats, such as ORC and Parquet, which support advanced compression techniques because the data of a column is stored together. Such techniques can reduce the file size by 70%.

There are several benefits of using a columnar file format. These include the following:

- In a columnar file format, data is more homogeneous since it is stored in the form of columns. Hence, it becomes easier to achieve high degrees of compression.
- IO is also reduced since we only need to scan a subset of a column.
- Data is homogenous and all of the data of a column is stored together. So, an encoding that is supported by modern processors can be used.

Serialization and Deserialization in Spark

Serialization is the process of converting the state of an object to bits and byte streams so that it can be either stored somewhere or transferred over a network.

Deserialization is the process of converting the byte stream back to the original object state.

Serialization plays an important role in the performance of Spark jobs. Any format that is slow to serialize objects into bits and byte streams, or consumes a large number of bytes, adversely affects the computation to a large extent.

In Python, you have access to the following two main types of Serializers: **Marshal Serializer** and **Pickle Serializer.**

Serialization is implemented for maintaining the performance of a job in distributed systems. It is especially helpful in storing objects on a disk or sending them over networks.

In the case of Spark RDDs, they may be serialized to:

- Decrease memory usage when stored in a serialised form;
- Reduce network bottleneck in processes such as shuffling (as the size of the data itself may be reduced); and
- Tune the performance of Spark operations, as this helps in reducing both Disk and Network IO.

Note that objects can be serialized before they are sent to Spark worker nodes.

In PySpark, serializers are set during the **creation of Spark Context**.

The table given below summarises the differences between the two types of serializers available in PySpark.

Marshal Serializer	**Pickle Serializer**
Marshal Serializer is much faster than Pickle Serializer in terms of rate of serialization and deserialization.	Pickle Serializer is much slower than Marshal Serializer in terms of rate of serialization and deserialization.
Mashal Serializer supports fewer data types.	Pickle Serializer supports nearly every type of Python object.

So, you need to choose a serializer based on your use case. Typically, if you want your Spark job to be more flexible, then you should select a Pickle Serializer. In contrast, if the number of data types that you need to support in your Spark job is limited and are supported by a Marshal Serializer, then you should prefer that since it is much faster than Pickle Serializer.

Spark Memory Management Parameters

Spark supports various memory levels. The RDD persistence logic that are to be followed while using memory levels are listed as follows:

- **MEMORY ONLY**: In this memory level, whenever an RDD has to be persisted, if the total size of the RDD is less than the Main memory, i.e., RAM, then it will persist in the memory; however, if the size of the RDD is greater than the size of the Main memory, then the spill-over partitions of the RDDs have to be processed every time you need to use the RDD.
- **MEMORY AND DISK**: This is similar to the MEMORY ONLY memory level, except the spill-over partitions of the RDD are persisted in the Disk if the RDD is bigger than the Main memory.
- **MEMORY-ONLY SERIALIZED**: This is almost similar to the MEMORY-ONLY memory level, except the RDD is stored as a serialized Java object before it is stored in the main memory. In this case, the RDDs are stored in a much more storage-efficient manner than the MEMORY-ONLY memory level.
- **MEMORY AND DISK SERIALIZED:** This is the same as the MEMORY AND DISK memory level. The only difference is that if the RDDs are big even after serialization, the spill-over partitions of the serialized RDDs are stored in the disk.
- **DISK ONLY:** In this memory level, RDDs are persisted directly in the Disk, not in the Main memory.
- The other two Memory levels, **MEMORY_ONLY_2** and **MEMORY_AND_DISK_2,** are almost the same as their normal counterparts. The only difference is that each partition of the RDDs is replicated on two cluster nodes.

Typically, if you need to store an RDD in the main memory itself, so that it does not have to be recalculated and processed each time, then you can do so using **Persist and Cache.**

If you fail to do this, then every time you call an **Action**, based on the lineage of the RDD, you will have to create the RDD from scratch; this would be time consuming and cause Disk IO.

Persist and cache simply store the intermediate computation so that it can be reused easily, thereby reducing the overall Disk IO and improving the job performance.

The difference between persist and cache is that cache stores RDDs in their default storage level, the MEMORY_ONLY level, and stores data sets in their default storage level, the MEMORY_AND_DISK level. On the other hand, you can use persist to assign user-defined storage levels (all memory levels are supported here) to RDDs and data sets. Both of these techniques are lazy in nature.

There is another concept, known as Checkpointing. Unlike cache and persist, checkpointing **breaks the Spark lineage** whenever it has to go to the previous stage. It is also computed **separately** from other jobs. Note that checkpointing is quite similar to checkpoints in the Windows operating system. This means you can set a checkpoint and later return to this checkpoint easily and reconfigure the machine if you need to in case some software update or installation has caused an error.

Checkpoint data is **persistent in nature** and is **not removed even after SparkContext is destroyed**. You will learn more about checkpointing in the subsequent modules on Spark Streaming.

You can use another technique, known as Unpersist. So far, you have learnt how to persist and cache your RDDs and data sets in the main memory. But if you need to persist some more data in the memory, then there might not be enough space left to persist the additional data.

If you want to remove these RDDs and data sets manually from the cache, then you can do so with the help of the Unpersist method. Even if you do not remove the data manually from the cache using Unpersist, cache follows the **Least Recently Used (LRU) principle** to automatically evict data.

Understanding Network IO

Typically, Network IO can be attributed to the time taken when data has to be sent between monitored processes. This means that whenever you have data that is present in, say, another remote machine or node, and is required for the current job, then the operation will be done to get the data over the network. That is what Network IO all about.

In Spark, Network IO occurs typically due to shuffles. Spark is a distributed processing system and so, data is distributed between a number of different machines and partitions. Shuffles basically refer to the phenomenon wherein data has to be rearranged between various partitions. A shuffle is essentially the movement of data across multiple partitions (which will mostly be sitting on a different node.).

Now, since you know that shuffles are the reason behind increase in Network IO, you will next learn at how to reduce shuffles:

- **Optimising joins:** Shuffles usually happen when a join operation occurs. Join is quite an expensive operation in data storage systems. In Spark, whenever a join occurs, data such as entire tables are moved across various partitions and, hence, optimising the techniques for joins will directly help in reducing shuffles.
- **Avoiding wide transformations:** Many wide transformation operations, such as groupByKey() and Sorting, involve the movement of a large amount of data across various partitions. If we can avoid such operations, we can avoid shuffles.

In any big-data-processing system, the overall volume of data is quite large and so, instead of bringing the data closer to the nodes where the algorithms and codes are located, you should try to bring the code to the partitions and nodes where the data is actually present. This whole concept is known as **Data Locality**. If you try to implement data locality in your Spark jobs, then it will try to keep the IO operations within a single physical node, which means you will be avoiding any IO operations, whatsoever, over the network.

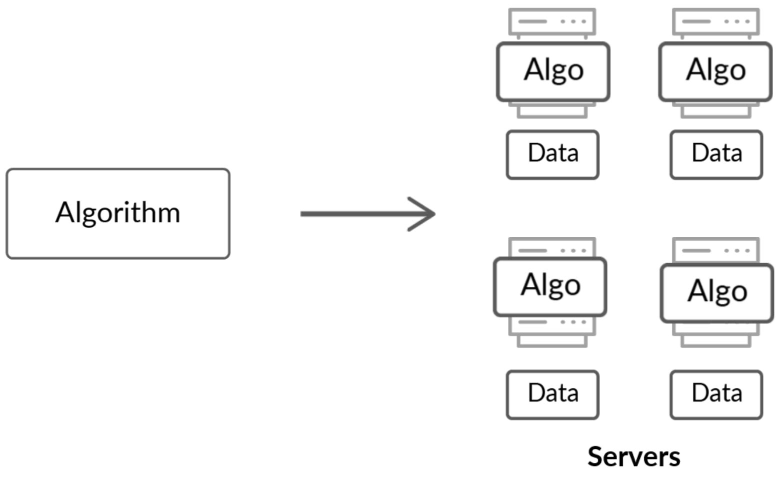


Figure 05: Data Locality

Just like there are various techniques to reduce Disk IO, there are several techniques to reduce Network IO as well. Most of these techniques directly affect shuffles and, thereby, help in reducing the overall Network IO. These techniques to reduce Network IO include the following:

- **Optimising partitioning techniques:** Using appropriate partitioning techniques will help you avoid unnecessary shuffles in your Spark jobs. In the case of joins, if you use the correct partitioning techniques, then similar data from two tables might be stored in the same machine, thereby reducing the shuffles required
- **Avoiding wide transformations:** Wide transformations are a form of transformation function that leads to the movement of data across several partitions. For example, if you want to perform a

groupByKey() operation, then all the different RDDs with the same key have to migrate and come to a single machine. This whole operation requires a lot of data shuffling. Hence, if wide transformations are absolutely required, then you should use alternative operations, such as **reduceByKey()**, which will try to reduce the data in the individual partitions before the actual shuffle happens.

- **Using broadcast joins:** Joins are expensive operations since, typically, a large amount of data has to be shuffled across partitions, so that data records with the same key are present in the same machines, so that the actual join operation can be performed. In some cases, where one table may be comparatively quite small compared with another table, you can use broadcast joins, which will basically broadcast this small table to all the partitions, thereby reducing shuffling quite effectively.

Understanding Shuffles

Shuffles in Spark refer to the **regrouping and redistribution of data** across various partitions and machines. Whenever you perform a **wide operation** in Spark, a shuffle takes place.

Some of the operations that lead to the occurrence of shuffles in Spark include the following:

- cogroup
- join
- groupByKey
- reduceByKey
- joinByKey
- sortByKey
- distinct
- intersection
- repartition
- coalesce

As you can see, all of these operations involve the movement of data between partitions (worker nodes) in one of the steps of their functions. Data has to be moved across machines so that it can be matched to its respective keys, in order to perform the particular functions, thus leading to a large amount of Network IO.

The nature of distributed systems means shuffles are bound to happen. However, too many shuffles would have a significant adverse effect on the performance of your Spark jobs.

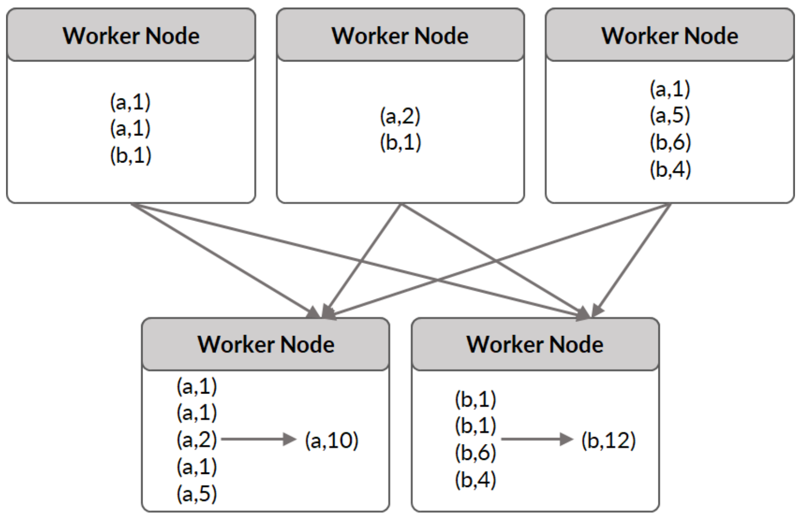


Figure 06: Process of Shuffles

As you can see in the image above, a shuffle is essentially a groupByKey operation after which the values of similar keys are added. As shown in the image, there are three worker nodes, which contain the data of two of the keys, a and b. For this operation, you need to get all the key–value pairs corresponding to one key on the same machine so that the sum can be computed. This movement of data is what is known as a shuffle.

So, now that you have learnt about shuffles, you will next learn about some methods to reduce shuffles:

- **Implementing optimal partitioning:** If you use optimal custom partitioning techniques, then a correct set of keys for partitioning would help you avoid a significant amount of shuffles in the data, since similar data is more likely to be partitioned together in the same nodes.
- **Using broadcast joins:** Joins are one of the most expensive operations in terms of Network IO, and at the same time, they are quite heavy in terms of shuffle operations. One of the methods with which you can reduce shuffles is that in cases where one of the tables is comparatively quite small, you can simply broadcast the smaller table to all the partitions containing the other tables.
- **Use reduceByKey() over groupByKey():** In cases where you need to use wide transformations, using more optimised operations, such as reduceByKey(), instead of the groupByKey() operation would help reduce shuffling to a great extent, since reduceByKey() results in comparatively much lesser Network IO overhead.

Optimising Joins in Spark

Joins are typically a process in which you try to get data from two different tables, and these tables are joined by some common columns. The tables may be present in two separate machines of a cluster. This naturally leads to the movement of data from one machine to another so that the join operation can be carried out. This is how joins lead to shuffling of data. At the industry level, especially in ETL applications, joins are not just one of the most common operations; they are also one of the heaviest in terms of compute load.

Traditionally, a Shuffle Hash join is what we normally use for joining two tables in Spark. This join not only involves shuffling a large amount of data, but you also need to create Hash tables, which makes the join process even more expensive.

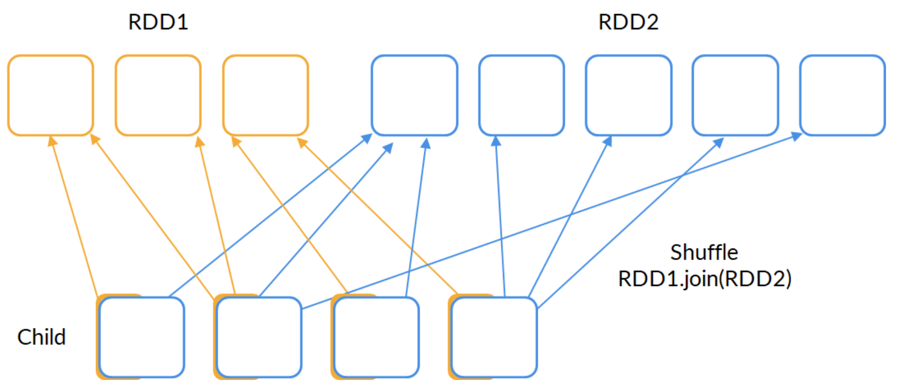


Figure 07: Shuffle Hash Join

As you can see in the image above, we have two RDDs, and when we try to join these RDDs, we need to find all the combinations between RDD1 and RDD2 which requires a lot of data to be shuffled across the different machines.

Shuffle Hash joins basically ensure that the data in each partition contains the same keys. It does so by partitioning the second table to be joined with the same partitioner as the first. Therefore, keys with the same hash value will be present in the same partition. Nevertheless, this process requires a lot of shuffles, as can be seen in the image above.

Spark supports almost all of the different types of joins that are there in a typical RDBMS, such as:

- Inner-Join,
- Left-Join,
- Right-Join,
- Outer-Join,
- Cross-Join,
- Left-Semi-Join and
- Left-Anti-Semi-Join.

In situations where you have to join two tables or RDDs, of which one of the tables or RDDs is quite big and the other one is relatively small in size, then instead of performing a blind join, which would again result in a large amount of shuffles, you can broadcast the smaller table to all the different partitions of the bigger table. What this does is that now you would not need any shuffling if you want to perform a join, and so, the original parallelism of the original table or RDD is maintained.

Typically, broadcast joins are quite useful in cases where the main data table has to be combined with a side table, for example, the metadata table.

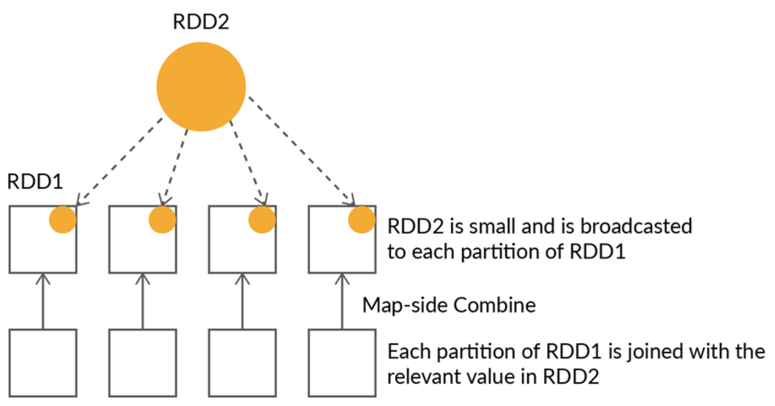

RDD2
RDD1
RDD2 is small and is broadcasted to each partition of RDD1
Map-side Combine
Each partition of RDD1 is joined with the relevant value in RDD2

Figure 08: Broadcast Join

Understanding Data Partitioning in Spark

Data partitioning is basically the process of dividing data into multiple parts. In distributed systems, whenever we need to process a single large file, we typically break it into smaller portions, and each of the small portions is called a partition. This same logic applies to data processing in Spark.

As the number of cores, Executor machines and partitions increases in a distributed system, there will be an increase in the degree of parallelism and so, the performance of a Spark job will be directly affected .

As can be expected, since data is divided into partitions, proper partitioning can help reduce Network IO:

- If you apply the correct type of partitioner and also have the optimal number of partitions, then shuffling wll be reduced drastically in case of wider transformations, such as groupByKey and reduceByKey, since there is a high possibility that similar data will be present on the same machines.
- You can also create your own custom partitioners, which would further help reduce the volume of data that is supposed to be shuffled, in case of shuffles.

Now, there are various operations that can benefit from the use of optimal partitioning techniques. These include the following:

- **Wide transformations, such as groupByKey() and reduceByKey():** As discussed previously, all of the different types of wide transformations, such as groupByKey(), will heavily benefit from the use of appropriate partitioning techniques in the performance of jobs, since shuffles will be reduced drastically.
- **Joins:** Since similar data will be present in the same partitions, it would help reduce the overall shuffle that is required in the process of joins.

Now, you will learn about some of the operations that can affect the partitions in a Spark job:

- **Repartitioning:** This method can be used to either increase or decrease the number of partitions in a Spark Job. Typically, whenever you call the Repartitioning method, the data is blindly shuffled across the machines to be allocated to new partitions. Generally, this method is used only when we need to increase the number of partitions.

- **Coalesce:** This method can be used to decrease the number of partitions in a Spark job. The difference between this method and Repartitioning is that coalesce will try to minimise the movement of data across partitions. Any unnecessary shuffle of data is avoided.

Let’s try and understand these two operations with the help of the example given below:

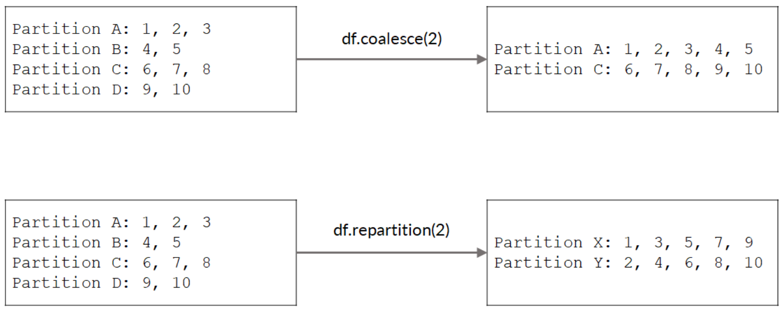


Figure 09: Coalesce and Repartition Example

As you can see in the example above, when we reduce the number of partitions with the help of coalesce, we retain the original Partitions A and C, and only data from Partitions B and D is redistributed among Partitions A and C. This ensures that there is minimal movement of data across the partitions.

If we compare this method to repartitioning using the same example, then you can see that all the values in the original partitions have been redistributed randomly into two partitions. Random shuffling has been done here and the values have been redistributed equally among the two resultant partitions.

Spark also allows you to create your own partitioner using the mechanism of Custom Partitioner wherein you can adjust the size and the number of partitions created, or even the partitioning scheme, according to the needs of your application.

While creating a Custom Partitioner, you should try to make it in such a way that the data is distributed more or less equally among all the partitions. Also, the number of partitions should be greater than the number of physical cores present in the cluster, to avoid underutilisation of cluster resources.

Why to Optimise Cluster Utilisation for Spark?

In Spark clusters, we typically have to take care of **Drivers** and **Executors** when we are trying to optimise our cluster utilisation. Both of these components have their own respective **memory**. In addition, Executors have their own **set of cores** on which the tasks run for your Spark jobs. This directly affects the degree of parallelism of your Spark jobs. Hence, we should ensure that these parameters are aligned with the requirements of our workload.

So far, you saw that most of our optimisations were **proactive** in nature, in that we were optimising our Spark jobs on our own. However, sometimes, especially in the case of cluster optimisation, our optimisations might need to be **reactive** in nature. For instance, if your Spark job is failing due to an OOM, or an **Out of memory error in your Driver logs**, then this means that the final result of your Spark job, which is ultimately supposed to be sent over to the Driver node, is much larger in size as compared with the Driver memory. In this case, you will need to **increase the Driver memory** in order to avoid this situation. Similarly, if you encounter this error in your **Executor logs**, then you may need to **increase your Executor memory**. You might also feel that your **job is running very slowly**. In this case, you might have to **increase the number of Executor cores** in order to increase parallelism and, thereby, increase the performance of your Spark cluster.

There exists an opposite case as well, wherein you may tune your parameters to be higher than required. If your **Driver memory is too large**, then it would lead to **wastage of resources**. Also, even though you might not encounter any out-of-memory error, you might still face issues with other jobs, since you might not have enough resources left for them. Similarly, your **Executor memory may also be very large**, which would again lead to **wastage of resources** and potential issues with other jobs.

The number of partitions that you have in your Spark cluster should also be more than the number of Executor cores in the cluster. This is to ensure that your Spark Executor nodes do not remain underutilised in any situation. However, you would again need to ensure that the number of partitions is not very high, as this will put stress on the Driver node as it will need to maintain the metadata of more partitions. Keeping too many Executor cores also leads to wastage of resources.

Next, you will learn how cluster utilisation can be improved:

- The size of Driver and Executor memory should always be based on the size of the data involved in a job. This will help optimise the performance of your jobs, while also preventing any cluster underutilisation. Each job may have different requirements and so, you cannot choose one particular configuration for all of your jobs.
- You should always try to strike the right balance between the performance benefits that you will reap from your cluster configuration and the specific costs that you will have to incur for using these services.
- A standard practise that you can follow while trying to choose the optimal configuration for your Spark cluster is to start with the default configuration for your Spark job and then try to alter the CPU and memory parameters accordingly. In many cases, your Spark job will perform well with the default configuration.

Job Deployment Modes in Spark

Spark provides various cluster deployment modes. These include the following:

Local mode: Local mode is a non-distributed, single-JVM deployment mode wherein all the components run on a single instance. Any job that you run locally on your own laptop or on any other machine is run in local mode. This mode is typically used only for testing, debugging and demonstration purposes, and not for actual ETL pipelines in the production environment. The degree of parallelism in this mode depends on the number of CPU cores that are present in your local machine. You can set this mode for the execution of your Spark jobs. A Spark session is started by specifying that your job is running in local mode in the master. An example for this is as shown below:

```
spark = SparkSession.builder.appName('demo').master("local").getOrCreate()
```

Standalone mode: Spark has the capability to manage clusters by itself. It uses its in-built Spark Resource Manager in the standalone mode. In this mode, Spark allows you to create a distributed Master–Slave architecture similar to that in Hadoop. By default, it is set to have a single-node cluster configuration, although it can be modified if needed.

Following is an example to see how you can use this mode:

```
spark-submit --master spark://207.184.161.138:7077
```

Note: You typically have to specify the IP address of the master node, followed by its port number, when you need to use this particular mode in Spark.

Spark typically has the following two different deployment modes:

- **Spark cluster mode:** If the Driver program resides on one of the worker nodes inside a cluster itself, then that mode of deployment is known as Spark Cluster Mode.
- **Spark Client Mode:** If the Driver program resides on an external client machine, then that mode of deployment is known as Spark Client Mode.

Note: In case of Cluster mode, the Spark driver can run on any of the available nodes along with the Spark Executor.

Since Spark is based on the Hadoop Ecosystem, it can also leverage **YARN, or Yet Another Resource Negotiator**, for cluster management, which, similar to Spark, has the following two deployment modes:

- **YARN client mode:** If the driver program resides on an external Client node, then the deployment mode is known as the YARN Client Mode. The image given below is an example of this deployment mode.

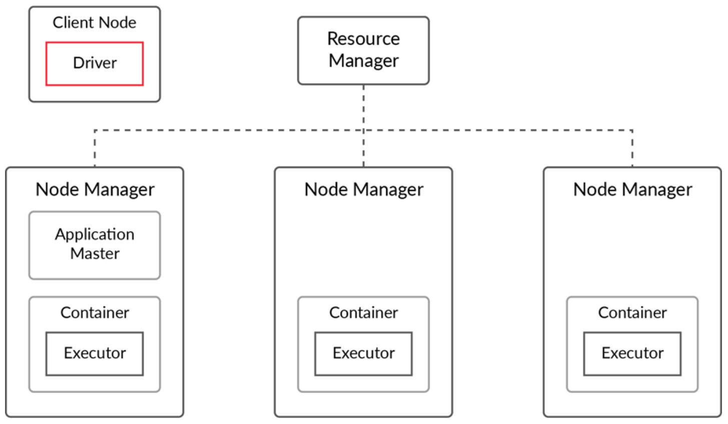


Figure 10: YARN Client Mode

- **YARN cluster mode:** If the Driver program resides on one of the worker nodes of a cluster under a separate Application Master, then the deployment mode is known as the YARN Cluster Mode. The image given below is an example of this deployment mode.

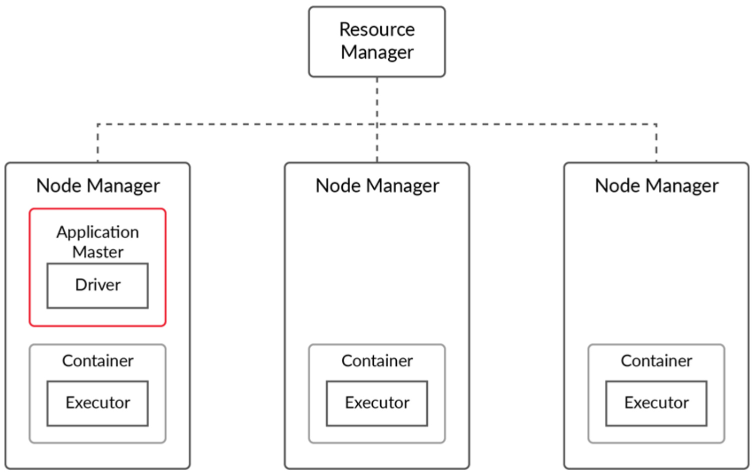
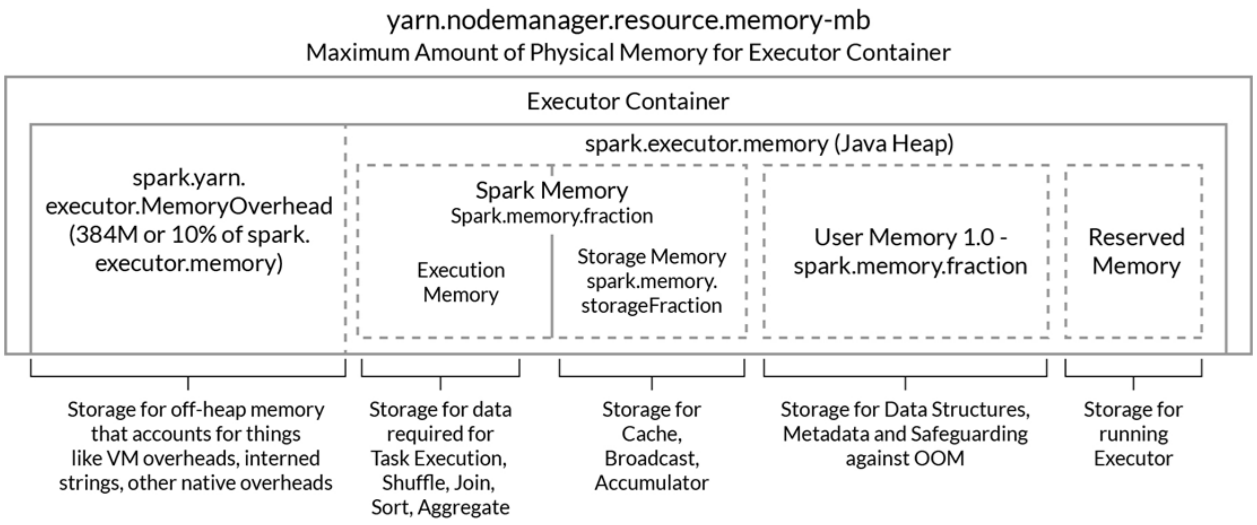


Figure 11: YARN Cluster Mode

Tuning Spark Memory and CPU Parameters

The Spark memory model for Executors has 10% reserved for memory overhead and the remaining 90% is divided among Spark Memory, User Memory and Reserved Memory.



yarn.nodemanager.resource.memory-mb
Maximum Amount of Physical Memory for Executor Container
Executor Container
spark.executor.memory (Java Heap)
spark.yarn. executor.MemoryOverhead (384M or 10% of spark. executor.memory)
Spark Memory
Spark.memory.fraction
Execution Memory
Storage Memory spark.memory. storageFraction
User Memory 1.0 - spark.memory.fraction
Reserved Memory
Storage for off-heap memory that accounts for things like VM overheads, interned strings, other native overheads
Storage for data required for Task Execution, Shuffle, Join, Sort, Aggregate
Storage for Cache, Broadcast, Accumulator
Storage for Data Structures, Metadata and Safeguarding against OOM
Storage for running Executor

Figure 12: Spark Memory Model for Executors

Spark Memory is further divided into Execution Memory and Storage Memory. Execution Memory is the storage for the execution of tasks, shuffles and other wide operations. Storage Memory is the memory for storing cache, Broadcast tables, etc.

User Memory is the memory that is reserved for storing data structures and metadata, and for the purpose of safeguarding against out-of-memory issues.

Finally, Reserved Memory is used for running the actual Executor node itself.

The main memory that is relevant to us is Spark Execution Memory, since this is the memory that decides the storage for the actual task execution and operations. Storage memory is also important as it stores cache and broadcast tables.

Spark contains various configuration parameters that can be modified. Some of the important ones are as follows:

- **spark.executor.memory:** Size of memory to use for each executor that runs a task
- **spark.executor.cores:** Number of virtual cores
- **spark.driver.memory:** Size of memory to use for the driver
- **spark.driver.cores:** Number of virtual cores to use for the driver
- **spark.executor.instances:** Number of executors. To be set unless spark.dynamicAllocation.enabled is set to true.
- **spark.default.parallelism:** Default number of partitions in RDDs returned by transformations such as join, reduceByKey and parallelise when the number of partitions is not defined by the user

The parameters above can be set in multiple ways. You can pass them in the program itself by passing the configuration during the initialisation of the Spark context in SparkConf(). You can also pass these configuration parameters in the spark-defaults.conf file, which will basically override the default parameters. Finally, there is another way to pass these parameters: passing them and running the job using the spark-submit command.

One point that you need to remember is that you should always try to run your Spark job once using the default parameters set in Spark Submit, since the majority of times, the default parameters are quite optimised for running most of your Spark jobs efficiently. To do this, you can simply remove all the options from the spark-submit command.

Another important point to consider is that you can always use the “spark-submit -h” command, discussed previously, to look at the various parameters that can be modified. Also, the AWS EMR documentation on Spark Submit has various formulae to benchmark the various options available in Spark submit if you want to modify your job even further.

Apache Spark in the Production Environment

There are several considerations that you need to make while running Spark jobs in the production environment.

Typically, version-control tools, such as Git, are used to keep track of the different versions of a Spark job, so that multiple developers can work simultaneously on the Spark job and you can go back to a previous version if needed.

A CI–CD pipeline is also set up in the production environment, as it reduces the effort and time required by a job after it has been written in the machine to be deployed into production. Standardised pipelines are constructed to ensure that this process is seamless.

Whenever you create a job and push the code, it results in the creation of a package or a bundled job. Each successful package creates a new version of a Spark job.

Spark jobs are typically executed as batch jobs and sometimes they have to be executed on-demand. Hence, appropriate hooks or a specialised scheduler framework is needed to run Spark jobs as a batch processing system.

Best Practices while Working with Apache Spark

While working with Apache Spark, there are several best practices that you can follow in order to optimise Spark jobs efficiently.

Typically, there is no single configuration that can satisfy the requirements of all types of Spark jobs. Different workloads can have different processing patterns and so, studying and analysing processing patterns can help you achieve the best results in terms of performance and efficiency.

You can always start from the default configuration and then monitor your parameters, and modify them accordingly to achieve better performance results. You should try to avoid making drastic changes initially and keep making incremental changes to improve the performance of our Spark jobs.

You should also consider the processing power of your Spark cluster and try to fully utilise its parallel processing capabilities. In this way, you can avoid underutilisation of your machines.

Also, you should try to keep the tasks of your Spark jobs small; a large size may result in a very large shuffle block, which may, in turn, lead to further errors. Also, large tasks will adversely affect the garbage collection process.

You also need to consider that your Spark job is one of the several jobs, such as the monitoring systems and other jobs, that are running parallely in your cluster. Hence, you must ensure that you do not use up all of the resources of the cluster.

Given below are brief descriptions of some common errors and solution approaches while working with Apache Spark. These were discussed in the video:

- **Driver OOM:** Increase Driver Memory
- **Executor OOM:** Increase Executor Memory
- **Too big a shuffle block:** Keep tasks small
- **Frequent Garbage collection:** Keep tasks small

Finally, Spark's official documentation is the ultimate resource whenever you have questions regarding any concept while optimising your Spark jobs.