Interview Questions - Optimising Spark for Large Scale Data Processing

1. **Why is parquet more optimised and space-saving than other row-level text formats such as CSV?**

To understand more about Parquet, you need to first understand the columnar format. In a column-oriented format, the values in each column of the same type are stored together.

For below row table like -

ID	Name	Phone
1	abc	9029
2	pqr	8902

Storage of the table above in row level format will be as follows -

```
1,abc,9029,2,pqr,8902
```

Storage of the table above in columnar format will be as follows -

```
1, 2,abc,pqr,9029,8902
```

Organising data by column allows better compression, as data of the same kind is always placed together. The space savings are very noticeable at the scale of a Hadoop cluster
Querying on any column is very efficient, as all the values of that column belong to the same place.

2. **In Spark, how can you identify if memory optimisation steps that you took, like broadcasting the variable/smaller DataFrames, are actually considered while running the job?**

   On any spark RDD/DF, you can simply run the explain() method. This helps to identify the actual logical and physical plan that spark is going to apply for the creation of that RDD. This can confirm whether the actual execution steps that Spark takes and your instructions are the same or not.

3. **What are the different approaches/ways in which you can optimise any Spark job?**

First is code optimisation, wherein:

- You broadcast the smaller DataFrames for reducing network calls,
- Use checkpoints to avoid failures and recreation of larger DataFrames, and
- Use caching to reuse the data frames from the memory itself.

Secondly, cluster optimisation, wherein:

- You select the right resources to be used from the cluster for the given Spark job,
- You properly partition the data so that it can be brought to a close locality like process or at least the node level

4. **What are the different debugging/troubleshooting steps that you can use to tackle spark memory errors, such as 'executor out of memory' or 'executor using more than its physical memory'?**

First, you analyse the data and check if any partition of data can be bigger than the spark executor memory. If yes, you repartition it in such a way that each partition is significantly smaller than the executor memory.

Then, analyse the Spark UI to identify the flow of the job and stages and check whether anything is wrong with the job flow, such as excessive shuffling steps.

Try to increase the executor memory just to identify how much memory it needs to fit the data partition and complete the job

5. **Why are serialization and deserialization techniques required in spark?**

Spark is a distributed processing framework. All the data that it processes has to travel through the network or saved in memory and hard disk.

For network travel and storage to HDD, the data needs to be converted into a byte stream. This conversion process is called serialization. Reading those streams back to the program is called deserialization.

A Spark program generally spends more time on these conversions. So, various techniques are created for various optimisations on these conversions.

6. **What is the difference between the persist() and cache() methods in Spark?**

Cache() is one of the versions of the persist() method. The persist() method can save the RDD in user-defined storage, such as the main memory or HDD. While cache() stores RDD in the main memory by default, cache() internally calls the persist() method for saving the RDD in the main memory.

7. **Can you think of a scenario in which you should use checkpoint() instead of persist()?**

- persist() saves the RDD and can reuse it multiple times. However, once the node or JVM fails, it loses track of the RDD and then the RDD needs to be created using a lineage graph. This means that Spark restarts the process from the beginning, that is, reading data from the data source and applying transformation on it.
- Incase reading data is costly from sources like S3 in terms of time taken as every point in data is an API call. In the case of JVM and node failures, reading data again from S3 might be a very costly factor if node failure is common in the cluster.
- In this case, checkpoint() is useful. It stores the actual data of the RDD and discards the lineage graph. This helps to restart the job from the point where it fails, thereby saving a lot of time.
- checkpoint() saves RDD in HDFS and can be read even if the node/JVM fails or is restarted.

8. **What is the most expensive operation In spark, and when is it needed?**

In any distributed processing engine, sending data across a network is the most expensive, as it involves serialization, disk I/O and Network I/O.
Shuffling is the step where data needs to be sent across the network.
Data shuffle is required when Wide operations and Join operations occur in a Spark job.

9. **What is the difference between groupbykey and reducebykey? Which of these is a more expensive operation?**

- groupbykey and reducebykey will fetch the same results. However, there is a significant difference in the performance of both functions. reduceByKey() works faster with large datasets than groupByKey().
- In reduceByKey(), data partitions on the same machine with the same key are combined before the data is shuffled. So, the data is already reduced at the node level. It is then shuffled where it is then reduced at the cluster level. This reduces the overall Network IO.
- In groupByKey(), all the key-value pairs are shuffled across the network. This is a lot of unnecessary data to transfer over the network. Hence, it is a more expensive operation than reduceByKey().

10. **Let's say we have two Spark data frames, one with 100 million records and another with 1,000 records, and we need to join the two. According to you, what is the best approach to doing this?**

As one DataFrame is very small, we will use Broadcast Join. In Broadcast Join, the smaller data set is sent across the network. So, while joining with larger data set partitions, the smaller data set is available in every node.

11. **Let's say you have an RDD of 10 terabytes, and you need to partition it. But the key of the RDD is in the form of numbers from the range (1, 10). Which partition technique will be useful here, and why?**

- As the number of keys is very less, if you use Hash partitions, then it will create 10 partitions, and each partition will be about 1 TB in size. So, a Hash partition is not a good choice here.
- You should go for a Range partition where a number of partitions will be created for each key, and they will be placed in sorted order. It will increase the number of partitions and also the performance of the Spark job

12. **Between Coalesce and Repartition, which has lower network shuffling? Which operations are recommended and in what situations?**

- Coalesce uses the existing partitions. So, it has less data shuffling in the network. It is recommended to use Coalesce if you want to reduce the number of partitions.
- Repartition uses network shuffling and recreates new partitions that are equal in size. It is recommended to use repartition when you want to increase the number of partitions.
- Coalesce is not recommended to increase the number of partitions, as it may create unequal size partitions, and a spark job does not work well with unequal size partitions. This may, in turn, create a need for network shuffling.

13. **Let's say you have a cluster of three worker nodes, each with three cores. In terms of parallelism, you have at most nine parallel tasks. Which of the following configurations is better in terms of Sark memory management?**

    1. **Number of executors = 3; each executor has 3 cores.**
    2. **Number of executors = 9; each executor has one core.**

You should definitely go with the first approach because when you increase the number of executors, it will increase the number of JVM instances.
Now, JVM has its own memory requirements, and caching is done at the executor level. So, each executor will try to cache some DataFrames that will increase the memory overlap and increase the RDD recreations.

So, it is recommended that each Executor should have more than one core so that some parallel tasks can be executed per JVM and also it should not be more than 4-5 as this will increase the burden of I/O on a single JVM.