# **Best Practices and Performance Tuning Activities for PySpark**

1. Initialize pyspark:

import findspark

findspark.init()

2.It should be the first line of your code when you run from the jupyter notebook. It attaches a spark to sys. path and initialize pyspark to Spark home parameter. You can also pass the spark path explicitly like below:

findspark.init(‘/usr/\*\*\*\*/apache-spark/3.1.1/libexec’)

This way the engine recognizes it as a spark job and sends it to the correct queue.

3.Create spark session with required configuration:

from pyspark.sql import SparkSession,SQLContext

sql\_jar="/path/to/sql\_jar\_file/sqljdbc42.jar"

spark\_snow\_jar="/usr/.../snowflake/spark-snowflake\_2.11-2.5.5-spark\_2.3.jar"

snow\_jdbc\_jar="/usr/.../snowflake/snowflake-jdbc-3.10.3.jar"

oracle\_jar="/usr/path/to/oracle\_jar\_file//v12/jdbc/lib/oracle6.jar"

spark=(SparkSession

.builder

.master('yarn')

.appName('Spark job new\_job')

.config('spark.driver.memory','10g')

.config('spark.submit.deployMode','client')

.config('spark.executor.memory','15g')

.config('spark.executor.cores',4)

.config('spark.yarn.queue','short')

.config('spark.jars','{},{},{},{}'.frmat(sql\_jar,spark\_snow\_jar,snow\_jdbc\_jar,oracle\_jar))

.enableHiveSupport()

.getOrCreate())

4.As a best practice, you should pass jar files for all the available database connections. This could be set either in the spark session or config file. This is because when you connect to an Oracle/SQL/snowflake database using the below code, you might get the “oracle.jdbc.driver.OracleDriver” class not found error if the engine picks an incorrect jar file.

data=spark.read.format("jdbc")

.option("url",tns\_path)

.option("dbtable",query)

.option("user",userid)

.option("password",password)

.option("driver","oracle.jdbc.driver.OracleDriver")

.load()

5.Use fetch size option to make reading from DB faster:

Using the above data load code spark reads 10 rows(or what is set at DB level) per iteration which makes it very slow when dealing with large data. When the query output data was in crores, using fetch size to 100000 per iteration reduced reading time 20-30 minutes. PFB the code:

data=spark.read.format("jdbc")

.option("url",tns\_path)

.option("dbtable",query)

.option("user",userid)

.option("password",password)

.option("fetchsize","100000")

.option("driver","oracle.jdbc.driver.OracleDriver")

.load()

6.Use batch size option to make writing to DB faster:

When the data was in crores, using batch size to 100000 per iteration reduced writing time 20-30 minutes. PFB the code:

data.write.format("jdbc")

.option("url",tns\_path)

.option("dbtable",schemaname.tablename)

.option("user",userid)

.option("password",password)

.option("fetchsize","100000")

.option("driver","oracle.jdbc.driver.OracleDriver")

.option("batchsize","100000")

.mode('append').save()

7.Cache/Persist Efficiently:

In the initial solution, it was fetching the data and doing serialization multiple times, and joining with the second table which results in a lot of iteration. This process was taking hours to complete initially.

Persist fetches the data and does serialization once and keeps the data in Cache for further use. So next time an action is called the data is ready in cache already. By using persist on both the tables the process was completed in less than 5 minutes. Using broadcast join improves the execution time further. We will be discussing that in later sections.

But you need to be careful while using persist. Overuse of persisting will result in a memory error. So keep clearing your data from memory when they are no longer used in the program.

8.Avoid using UDF functions unless that is the only option:

User-defined functions de-serialize each row to object, apply the lambda function and re-serialize it resulting in slower execution and more garbage collection time.

9.Use of Thread wherever necessary:

If there are multiple independent actions in one job, you can use a thread to call those actions simultaneously. For example, in one job we were reading many huge tables from one schema and writing to another schema. Due to sequential action, the job was taking more than 2 hours. After we used the thread for concurrent writing, the load time was reduced to 30 minutes. Please note you might need to increase the spark session configuration. For optimum use of the current spark session configuration, you might pair a small slower task with a bigger faster task.

Use mapPartitions() inste

ad of map():

10.Both are rdd based operations, yet map partition is preferred over the map as using mapPartitions() you can initialize once on a complete partition whereas in the map() it does the same on one row each time.

11.Miscellaneous:

Avoid using count() on the data frame if it is not necessary. Remove all those actions you used for debugging before deploying your code.

Write intermediate or final files to parquet to reduce the read and write time.

If you want to read any file from your local during development, use the master as “local” because in “yarn” mode you can’t read from local.

In yarn mode, it references HDFS. So you have to get those files to the HDFS location for deployment.