Performance Troubleshooting – Diagnostic Actions

This section outlines the concrete actions I would take to investigate Spark job performance bottlenecks.

Key Diagnostic Questions

1.Are tasks stuck on input/output operations?

2. Are there too many partitions or too few?

3.Are some tasks significantly slower than others?

4.Is the job performing expensive shuffles?

5.Is garbage collection taking >10% of executor time?

Corrective Actions

1. It means tasks are waiting too long to

- Read input data
- Write output data,
- Shuffle intermediate data.

Symptoms like, taks with long duration but low CPU time, executors doing little processing but taking a long time or spark UI shows dominating I/O wait dominating task time.

- We can use parquet over CSV or JSON and a snappy compression (.option("compression", "snappy")).
- Utilize df.coalesce(50).write.parquet(), in order to reduce number of output files or, df.write.option(“maxRecordsPerFile”,1_000_000).parquet().
- Repartition before shuffle df = df.repartition("join_key")
- Using cache() wisely, then unpersist it

2. Too few or too many partitions (df.rdd.getNumPartitions()), for the first one we are underutilizing the cluster, causing long tasks, no parallelism. In contrast, for the second one we cause overhead, too many tasks, high GC.

So we can use df = df.repartition(200), what improves parallelism and triggers a full shuffle or df.coalesce(100), which doesn’t shuffle and it’s good before writing, because avoid tiny files.

In Addition AQE in Spark can handle partitioning dynamically at runtime. Helping in coalescing shuffle partitions automatically, switching join strategies (from sort-merge to broadcast), skewing join handling (adaptive salting-like logic)

spark.conf.set("spark.sql.adaptive.enabled", True) # usually True by default

4. To detect expensive shuffles I use df.explain(True) and look for SortMergeJoin for instance. In spark UI → DAG visualization, you can navigate through jobs and stages and check “Exchange” tiles in order to see duration, kind of wide transformation, etc.

If it’s possible, utilize Broadcast join, but the df should fit into memory .
For shuffle-heavy wide stages reduce df.select() columns.
Data skew in joins, apply salting or repartition(“key”), spark.speculation

5. If GC time is high, I will avoid df.collect() or df.toPandas(), the first one brings the entire dataframe to driver and the second one converts it to pandas, both creates large objects in memory, leading to:

- Out of memory
- High GC time
- Long serialization/deserialization

Instead .limit().collect(), show(), .take(n) for debugging, otherwise just apply collect() for small, filtered or aggregated data.

Same with df.cache(), only when is necessary. When you know you're going to use it multiple times and always df.unpersist() afterwards.

If available, I would also export Spark metrics to InfluxDB and visualize them in Grafana for time-series analysis.

Also there is a Scala listener "SparkMeasure" , packaged to work with Python.
we can import it and printout reports

Example:

```
from sparkmeasure import StageMetrics

spark = SparkSession.builder.appName("SparkMeasureDemo")
.config("spark.jars.packages", "ch.cern.sparkmeasure:spark-measure_2.12:0.23")
.getOrCreate()

stagemetrics = StageMetrics(spark)
stagemetrics.begin()

df = spark.range(0, 10000000).withColumn("mod", col("id") % 5)

result = df.groupBy("mod").count().collect()

stagemetrics.end()

stagemetrics.print_report()
```