9:) How to process 1 TB of data in Spark!

Here this 1 TB data is stored in HDFS, in HDFS the Block size is = 128 MB

$1TB = (1 * 1024 * 1024) / 128 = 8192$ Blocks

Let us take

→ 20 Executors
→ 5 cores in each executor
→ 6 GB RAM in each executor

And the cluster can perform (20 × 5 = 100) parallel tasks. Here tasks mean block so 100 blocks can be processed parallely.

$100 * 128 MB = 12800 MB / 1024 = 12.5 GB$

(12 GB data will get processed in 1st batch)

since the RAM size in 6 GB in each executor

(20 Exe * 6 GB = 120 GB Total RAM)

So at a time 12 GB of RAM will be occupied in the cluster.

20 Node / 12 GB = 1.6 GB RAM from each executor will be in use.

Now, Available RAM in each executor (6GB - 1.6GB = 4.4GB) will be free to use by other user's job.

So, 1TB = 1024GB / 12GB
= 85 Batches

Whole data will get processed in around 85 Batches

Failure Scenarios in Spark

① Out of Memory Error

- Increase executor memory
- Check if broadcast variable is Big in size
- Avoid unnecessary data shuffle
- If driver is out of memory then check

if collect() action was called

② Network Error: It happens when spark driver & executor are not able to communicate with each other.

Solution { spark.network.timeout
spark.executor.heartbeatinterval

③ Data Input/Output

↳ Make sure data is consistent as per data contract
↳ Always put read/write related operations in try-except block for easy debugging.

④ Serialization Error: Data is not according to the serialized schema.

⑤ Disk Space Error: Spark job spills intermediate data into the local

machine disk. In no sufficient memory is there then job will fail.

Solution: spark.local.dir

⑥ Memory Configuration

If we are facing heap space related issues and memory overhead issues then try to configure memory components effectively.

→ Analyze the data volume.

→ Define driver & executor memory properly.

→ Adjust memory configuration params properly.

Frequently used optimizations in Spark

① Partitioning:

↳ repartition() → Increase / Decrease

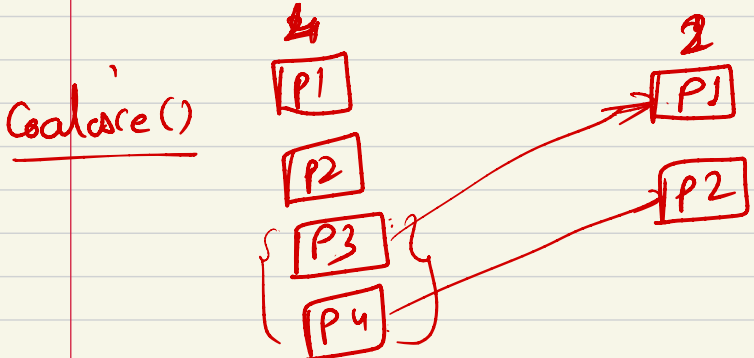
↳ Coalesce() → Decrease

empDF = spark.read_csv("employee.csv")

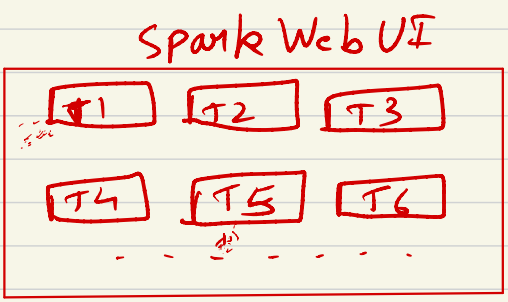
We are reading from HDFS, so let say 20 Block = 20 partitions.

In the code itself, we want to increase the number of partitions we can use repartition().

empDF.repartition(40)



empDF.coalesce(1).write( )



joinDF.repartition(100)

② Optimization in case of skewness

We can use Key-Salting technique.

③ Broadcast variables: It will help to broadcast dataset on multiple executors.

④ Persist() and Cache():

- Cache() → only in memory save
- Persist() → Memory + Disk both

⑤ Try to minimize number of stages in the spark job. That means unoptimized use of wide transformation.

---

* Difference between Job vs Stage vs Task?

Job → As soon as we call an action in spark application, new Job will be created.

Stage → Wherever data shuffling takes place in the spark app, a new stage will get created

Task → Small unit of execution.

empDF → (1)

deptDF → (2)

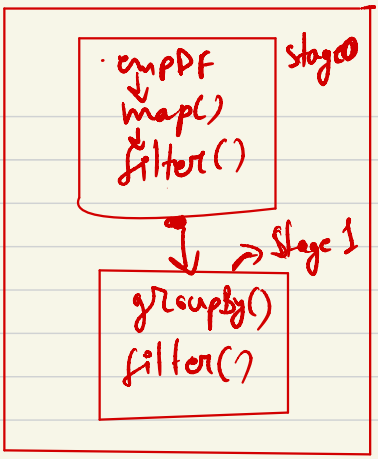
empDF.map()
empDF.filter('INDIA')] → Stage 0

Trans { empDF.groupBy('dept_id').sum(salary) → Stage 1
empDF.filter("Salary_sum < 50000")]

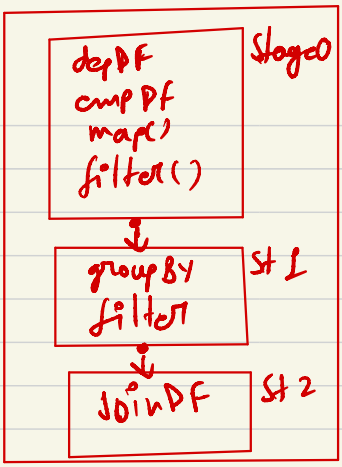
Action { empDF.collect(5)

Trans { joinDF = empDF.join(deptDF)
joinDF.filter(deptDF.dept_id not null)

Action { joinDF.collect(10)


empDF
map()
filter()
Stage 0
Stage 1
groupBy()
filter()

Job 0


depDF
empDF
map()
filter()
Stage 0
groupBy
filter
St 1
joinDF
St 2

Job 1