To prevent **memory spill** and **disk spill** in Spark jobs, especially on a large cluster, you need to optimize resource usage, memory allocation, partitioning, and shuffle operations. Memory and disk spills happen when Spark cannot fit intermediate data in memory, forcing it to write to disk, which slows down job performance.

### **1. Common Causes of Memory and Disk Spill**

1. **Insufficient Executor Memory**:
   * Executors run out of memory for intermediate computations.
2. **Improper Partitioning**:
   * Data skew or too few partitions can overload individual tasks.
3. **Large Shuffle Operations**:
   * High shuffle volumes exceed the memory allocated for shuffle buffers.
4. **Suboptimal Configurations**:
   * Misconfigured memory settings or task parallelism.
5. **Inefficient Data Processing**:
   * Operations like groupByKey or reduceByKey create large intermediate datasets.

### **2. Strategies to Prevent Memory and Disk Spill**

#### **a) Optimize Executor Memory Allocation**

* Allocate sufficient memory to executors to handle large datasets.

Adjust the executor memory based on cluster capacity:  
bash  
Copy code  
--executor-memory 8G

Enable off-heap memory for better memory management:  
bash  
Copy code  
--conf spark.memory.offHeap.enabled=true

--conf spark.memory.offHeap.size=2G

#### **b) Increase Spark Memory Fractions**

Tune memory fractions to allocate more memory for execution and storage:  
bash  
Copy code  
--conf spark.memory.fraction=0.8

--conf spark.memory.storageFraction=0.3

* + spark.memory.fraction: Portion of executor memory used for execution and storage.
  + spark.memory.storageFraction: Portion of memory reserved for caching.

#### **c) Avoid Large Shuffle Operations**

Replace **wide transformations** like groupByKey with more memory-efficient operations like reduceByKey or aggregateByKey:  
python  
Copy code  
rdd.reduceByKey(lambda x, y: x + y)

Enable shuffle compression to reduce data size:  
bash  
Copy code  
--conf spark.shuffle.compress=true

--conf spark.shuffle.spill.compress=true

Use **Broadcast Joins** for small datasets to reduce shuffle overhead:  
python  
Copy code  
broadcasted = broadcast(small\_df)

result = large\_df.join(broadcasted, "key")

#### **d) Proper Partitioning**

Repartition large datasets to distribute data evenly and avoid data skew:  
python  
Copy code  
rdd = rdd.repartition(2000) # Increase number of partitions

For shuffle operations, adjust the number of partitions:  
bash  
Copy code  
--conf spark.sql.shuffle.partitions=2000

#### **e) Balance Tasks and Executors**

* Ensure the number of tasks matches the cluster's resources:
  + Number of tasks ≥ total cores.

For example, with 500 cores:  
bash  
Copy code  
--conf spark.default.parallelism=500

#### **f) Monitor and Mitigate Data Skew**

* Identify skewed keys that cause uneven partition sizes and memory pressure.

Add **key salting** to distribute skewed keys across multiple partitions:  
python  
Copy code  
from pyspark.sql.functions import lit, concat, rand

df = df.withColumn("salted\_key", concat(df["key"], lit("\_"), (rand() \* 10).cast("int")))

#### **g) Enable Adaptive Query Execution (AQE)**

Dynamically optimize shuffle partitions and execution plans:  
bash  
Copy code  
--conf spark.sql.adaptive.enabled=true

--conf spark.sql.adaptive.coalescePartitions.enabled=true

--conf spark.sql.adaptive.shuffle.targetPostShuffleInputSize=64MB

#### **h) Cache Data Wisely**

* Avoid excessive caching or caching large datasets unnecessarily.

Cache only the required portions:  
python  
Copy code  
df.cache()

Use efficient storage levels for caching:  
python  
Copy code  
df.persist(StorageLevel.MEMORY\_AND\_DISK)

#### **i) Optimize Storage and Disk I/O**

* Use faster storage (e.g., SSDs) for Spark scratch space.

Configure Spark's local directories for spilling:  
bash  
Copy code  
--conf spark.local.dir=/mnt/ssd/spark

### **3. Key Configuration Parameters**

| **Parameter** | **Description** | **Recommended Value** |
| --- | --- | --- |
| spark.memory.fraction | Fraction of JVM heap reserved for execution and storage. | 0.8 |
| spark.memory.storageFraction | Fraction of memory reserved for storage (caching). | 0.3–0.4 |
| spark.sql.shuffle.partitions | Number of shuffle partitions. | Depends on dataset size (e.g., 2000). |
| spark.shuffle.compress | Enables shuffle compression to reduce disk and network I/O. | true |
| spark.shuffle.spill.compress | Enables spill compression for shuffle operations. | true |
| spark.memory.offHeap.enabled | Enables off-heap memory allocation. | true |
| spark.memory.offHeap.size | Amount of off-heap memory to allocate. | 2G (or based on job requirements). |
| spark.local.dir | Directory for temporary storage during spills. | Use high-speed storage (e.g., SSD). |
| spark.sql.adaptive.enabled | Enables Adaptive Query Execution (AQE). | true |
| spark.sql.adaptive.shuffle.targetPostShuffleInputSize | Target size for shuffle partitions post-AQE optimization. | 64MB |

### **4. Monitoring and Troubleshooting**

1. **Spark UI:**
   * Check the **Stages** tab for task execution times and spilled data.
   * Look at **Shuffle Read/Write Metrics** to identify problematic stages.
2. **Driver and Executor Logs:**
   * Look for warnings like Spilling data to disk or MemoryStore dropped....
3. **Cluster Monitoring Tools:**
   * Use tools like Ganglia, Prometheus, or Cloudera Manager to monitor resource usage.

### **5. Practical Example Configuration**

#### **Scenario:**

* Input data size: **500 GB**.
* Cluster with 100 cores and 1 TB memory.

#### **Spark Submit Command:**

bash

Copy code

spark-submit \

--master yarn \

--deploy-mode cluster \

--executor-memory 8G \

--executor-cores 4 \

--num-executors 25 \

--conf spark.memory.fraction=0.8 \

--conf spark.memory.storageFraction=0.3 \

--conf spark.sql.shuffle.partitions=2000 \

--conf spark.shuffle.compress=true \

--conf spark.shuffle.spill.compress=true \

--conf spark.sql.adaptive.enabled=true \

--conf spark.sql.adaptive.coalescePartitions.enabled=true \

--conf spark.local.dir=/mnt/ssd/spark \

my\_spark\_job.py

### **6. Best Practices**

1. **Right-Size Executors:**
   * Keep executor memory and cores proportional to workload.
2. **Optimize Partitioning:**
   * Ensure even distribution of data across partitions.
3. **Leverage AQE:**
   * Use Adaptive Query Execution to handle dynamic workloads efficiently.
4. **Monitor Job Performance:**
   * Regularly analyze the Spark UI to identify and address bottlenecks.
5. **Tune Incrementally:**
   * Start with default configurations and incrementally adjust based on job behavior.

Would you like assistance with a specific Spark job or configuration?

4o

##### **You said:**

When you optimize the spark job you need to consider spark.sql.shuffle.partitions or dynamic executor allocation which one will overwrite

###### **ChatGPT said:**

ChatGPT

When optimizing a Spark job, **spark.sql.shuffle.partitions** and **dynamic executor allocation** serve different purposes, and they do not directly overwrite each other. However, their interaction can influence performance significantly. Here’s a detailed explanation of their roles and how they interact:

### **1. Role of spark.sql.shuffle.partitions**

* **Purpose:**
  + Controls the number of partitions created during shuffle operations (e.g., join, groupBy, repartition).
  + Determines how data is divided among tasks in shuffle stages.
* **Default Value:**
  + The default is 200, which may not be optimal for large datasets or clusters with many executors.
* **Effect on Performance:**
  + **Too Few Partitions:** Leads to underutilization of cluster resources and larger partitions, increasing the chance of memory and disk spills.
  + **Too Many Partitions:** Causes overhead due to excessive task creation and smaller shuffle sizes, increasing scheduling overhead.

### **2. Role of Dynamic Executor Allocation**

* **Purpose:**
  + Dynamically adjusts the number of executors in response to the workload.
  + Adds executors when more tasks need to run and removes idle executors to free up cluster resources.
* **Effect on Performance:**
  + Ensures efficient resource usage by scaling the cluster size dynamically.
  + Prevents over-provisioning or underutilization of cluster resources.

### **3. Interaction Between spark.sql.shuffle.partitions and Dynamic Executor Allocation**

#### **Does One Overwrite the Other?**

No, they **do not overwrite** each other. However:

* **spark.sql.shuffle.partitions** determines the number of tasks created in a shuffle stage.
* **Dynamic Executor Allocation** determines how many executors are available to process those tasks.

### **4. Key Interactions to Consider**

#### **a) Impact of Shuffle Partitions on Dynamic Executor Allocation**

* If the number of shuffle partitions is **too low**, tasks are fewer than the available cores, causing underutilization even if more executors are dynamically added.
  + Example:  
    If spark.sql.shuffle.partitions = 200 and the cluster has 500 cores, only 200 tasks will run, leaving 300 cores idle.
* If the number of shuffle partitions is **too high**, many small tasks are created, which increases task scheduling overhead and may require more executors than necessary.

#### **b) Impact of Dynamic Executor Allocation on Shuffle Partitions**

* Dynamic executor allocation adjusts the number of executors based on the workload, but it does not change the number of partitions during a shuffle.
* Tasks created from the shuffle partitions are distributed across the available executors. If the executor count is low, tasks are queued and take longer to complete.

### **5. How to Optimize Both Together**

To achieve the best performance, consider these steps:

#### **a) Optimize spark.sql.shuffle.partitions**

1. **Based on Input Data Size:**
   * Calculate partitions so that each partition holds approximately **128 MB–256 MB** of data.

python  
Copy code  
num\_partitions = total\_data\_size / target\_partition\_size

1. Example:

For a 1 TB dataset and 128 MB partitions:  
makefile  
Copy code  
num\_partitions = 1 TB / 128 MB = 8192

1. **Based on Cluster Size:**
   * Ensure spark.sql.shuffle.partitions is proportional to the total cores available in the cluster.

bash  
Copy code  
num\_partitions >= num\_executors \* executor\_cores

#### **b) Configure Dynamic Executor Allocation**

1. **Set Minimum and Maximum Executors:**
   * Configure spark.dynamicAllocation.minExecutors and spark.dynamicAllocation.maxExecutors based on cluster resources.

bash  
Copy code  
--conf spark.dynamicAllocation.minExecutors=10

--conf spark.dynamicAllocation.maxExecutors=100

1. **Enable External Shuffle Service:**
   * Required for dynamic allocation to persist shuffle data when executors are removed.

bash  
Copy code  
--conf spark.shuffle.service.enabled=true

1. **Monitor Task Completion Times:**
   * Use Spark UI to adjust settings for efficient task execution.

### **6. Example Configuration**

#### **Scenario:**

* Input data size: **500 GB**.
* Cluster with **50 executors**, each having **4 cores** and **8 GB memory**.

#### **Configuration:**

bash

Copy code

spark-submit \

--master yarn \

--deploy-mode cluster \

--executor-memory 8G \

--executor-cores 4 \

--num-executors 50 \

--conf spark.dynamicAllocation.enabled=true \

--conf spark.dynamicAllocation.minExecutors=10 \

--conf spark.dynamicAllocation.maxExecutors=50 \

--conf spark.sql.shuffle.partitions=2000 \

--conf spark.shuffle.service.enabled=true \

my\_spark\_job.py

* **Explanation:**
  + spark.sql.shuffle.partitions=2000: Distributes shuffle tasks evenly for the dataset size and cluster resources.
  + Dynamic allocation adjusts the number of executors based on the workload.

### **7. When to Prioritize Each**

| **Scenario** | **Prioritize** |
| --- | --- |
| Small dataset or single-job cluster | Focus on spark.sql.shuffle.partitions. |
| Multi-tenant cluster with varying workloads | Use **dynamic executor allocation** to share resources efficiently. |
| Large datasets with heavy shuffle operations | Tune both to ensure efficient partitioning and dynamic scaling. |

### **8. Best Practices**

1. **Monitor Jobs:**
   * Use Spark UI to monitor task distribution, shuffle metrics, and executor utilization.
2. **Tune Incrementally:**
   * Start with defaults and incrementally adjust spark.sql.shuffle.partitions and dynamic allocation settings.
3. **Test Scaling:**
   * Test how the cluster scales with spark.dynamicAllocation.maxExecutors to ensure tasks are not bottlenecked.
4. **Balance Partitioning:**
   * Ensure spark.sql.shuffle.partitions aligns with the cluster's size and workload to avoid underutilization or overhead.

Would you like further assistance with configuring these parameters for a specific job?

4o

##### **You said:**

yes

###### **ChatGPT said:**

ChatGPT

Please share the specifics of your Spark job or workload, such as:

1. **Input Data Characteristics:**
   * Size of the input data (e.g., 100 GB, 1 TB).
   * Type of data source (e.g., HDFS, S3, database, Parquet files).
2. **Job Type:**
   * Heavy shuffle operations (e.g., joins, groupBy, aggregations).
   * Computational operations (e.g., transformations, machine learning).
3. **Cluster Configuration:**
   * Number of nodes, cores per node, and total memory.
   * Are other jobs running on the same cluster?
4. **Existing Configuration:**
   * Current settings for spark.sql.shuffle.partitions, dynamic executor allocation, executor memory, and cores.
5. **Performance Issues:**
   * Are you experiencing memory or disk spills, slow stages, or task failures?

With these details, I can provide tailored configuration recommendations to optimize your Spark job.

4o

##### **You said:**

Inut size of data 1 TB and number of nodes 5 and each of them have 800GB memory and 90 cores and we need to join 1 TB data coming from oracle database with hdfs 1TB data and we need to enable dynamic executor and need to configure spark.sql.shuffle.partitions

###### **ChatGPT said:**

ChatGPT

To efficiently handle **joining 1 TB of data from an Oracle database** with **1 TB of data from HDFS** using Spark on a cluster with **5 nodes (each with 800 GB memory and 90 cores)**, here’s how you can configure **dynamic executor allocation** and **spark.sql.shuffle.partitions** for optimal performance:

### **1. Key Considerations**

* **Data Size:** Total data size = **2 TB** (1 TB from Oracle + 1 TB from HDFS).
* **Cluster Resources:**
  + Total cores = 5×90=4505 \times 90 = 4505×90=450
  + Total memory = 5×800 GB=4,000 GB5 \times 800 \, \text{GB} = 4,000 \, \text{GB}5×800GB=4,000GB
* **Join Operation:**
  + Joins require a shuffle stage, which produces intermediate data.
  + Proper partitioning and dynamic executor allocation are crucial to minimize shuffle overhead and ensure efficient resource utilization.

### **2. Recommended Configuration**

#### **a) Dynamic Executor Allocation**

Enable dynamic allocation to efficiently scale the number of executors based on the workload.

bash

Copy code

--conf spark.dynamicAllocation.enabled=true

--conf spark.dynamicAllocation.minExecutors=10

--conf spark.dynamicAllocation.maxExecutors=100

--conf spark.dynamicAllocation.initialExecutors=20

--conf spark.dynamicAllocation.executorIdleTimeout=60s

--conf spark.shuffle.service.enabled=true

* **Reasoning:**
  + **Min Executors (10):** Start with enough executors to handle initial stages of the job.
  + **Max Executors (100):** Allow the job to scale to handle shuffle stages efficiently.
  + **Initial Executors (20):** Provide a good starting point for the job before scaling dynamically.
  + **Idle Timeout (60s):** Free idle executors quickly to prevent resource wastage.

#### **b) Shuffle Partitions**

Set spark.sql.shuffle.partitions based on the data size and number of executors.

1. **Partition Size Rule of Thumb:**
   * Target partition size: **128 MB–256 MB**.
   * Total shuffle partitions: num\_partitions=Total Data SizePartition Size\text{num\\_partitions} = \frac{\text{Total Data Size}}{\text{Partition Size}}num\_partitions=Partition SizeTotal Data Size​ For 2 TB2 \, \text{TB}2TB data and 128 MB128 \, \text{MB}128MB partitions: num\_partitions=2 TB128 MB=16,384 partitions.\text{num\\_partitions} = \frac{2 \, \text{TB}}{128 \, \text{MB}} = 16,384 \, \text{partitions}.num\_partitions=128MB2TB​=16,384partitions.
2. **Partitioning for Cluster Resources:**
   * Ensure at least 2–4 tasks per core to fully utilize the cluster.
   * For 450 cores450 \, \text{cores}450cores: Recommended Partitions=max⁡(16,384, 450×4)=16,384.\text{Recommended Partitions} = \max(\text{16,384}, \, 450 \times 4) = 16,384.Recommended Partitions=max(16,384,450×4)=16,384.

Set:  
bash  
Copy code  
--conf spark.sql.shuffle.partitions=16384

#### **c) Executor Configuration**

Balance executor memory and cores to optimize task execution.

1. **Executor Cores:**
   * Use 4 cores per executor4 \, \text{cores per executor}4cores per executor to allow parallelism while avoiding task starvation.
   * Total executors = Total CoresCores per Executor=4504=112.5 executors\frac{\text{Total Cores}}{\text{Cores per Executor}} = \frac{450}{4} = 112.5 \, \text{executors}Cores per ExecutorTotal Cores​=4450​=112.5executors.
2. **Executor Memory:**
   * Allocate 40 GB40 \, \text{GB}40GB memory per executor to ensure sufficient memory while avoiding excessive garbage collection.
   * Reserve space for overhead and YARN (e.g., 7%–10%): Usable Memory per Executor=40 GB (excluding overhead)\text{Usable Memory per Executor} = 40 \, \text{GB} \, (\text{excluding overhead})Usable Memory per Executor=40GB(excluding overhead)

**Configuration:**bash  
Copy code  
--conf spark.executor.cores=4

--conf spark.executor.memory=40G

#### **d) Broadcast Joins**

If one of the datasets (e.g., the Oracle data) is small enough to fit in memory, use **broadcast joins** to avoid shuffle for that dataset.

python

Copy code

from pyspark.sql.functions import broadcast

oracle\_df = spark.read.format("jdbc").option("url", "<oracle\_url>").option("dbtable", "table\_name").load()

hdfs\_df = spark.read.parquet("hdfs\_path")

result = hdfs\_df.join(broadcast(oracle\_df), "key\_column")

**Broadcast Threshold:**bash  
Copy code  
--conf spark.sql.autoBroadcastJoinThreshold=10MB

### **3. Example Spark Submit Command**

bash

Copy code

spark-submit \

--master yarn \

--deploy-mode cluster \

--conf spark.dynamicAllocation.enabled=true \

--conf spark.dynamicAllocation.minExecutors=10 \

--conf spark.dynamicAllocation.maxExecutors=100 \

--conf spark.dynamicAllocation.initialExecutors=20 \

--conf spark.shuffle.service.enabled=true \

--conf spark.executor.cores=4 \

--conf spark.executor.memory=40G \

--conf spark.sql.shuffle.partitions=16384 \

--conf spark.sql.autoBroadcastJoinThreshold=10MB \

my\_spark\_job.py

### **4. Monitoring and Optimization**

#### **a) Use Spark UI**

* Check:
  + **Task Duration:** Look for stragglers or tasks taking disproportionately long.
  + **Shuffle Read/Write:** Ensure partitions are balanced.
  + **Executor Utilization:** Verify that executors are fully utilized.

#### **b) Adjust Based on Observations**

* **Increase Partitions:** If shuffle tasks are too large or causing memory spills.
* **Decrease Partitions:** If there are too many small tasks causing scheduling overhead.

### **5. Summary**

| **Configuration** | **Recommended Value** | **Reasoning** |
| --- | --- | --- |
| **Dynamic Executors** | Min: 10, Max: 100, Initial: 20 | Efficient scaling for workload dynamics. |
| **Shuffle Partitions** | spark.sql.shuffle.partitions=16384 | Optimized for 2 TB data with 128 MB partitions. |
| **Executor Cores** | spark.executor.cores=4 | Balances task parallelism and resource utilization. |
| **Executor Memory** | spark.executor.memory=40G | Prevents memory spills while avoiding GC overhead. |
| **Broadcast Joins** | Enable for small datasets (<10 MB) | Avoids shuffle for smaller datasets. |

Would you like help implementing this setup or analyzing the performance?

4o

##### **You said:**

IN the spark job executor jvm takes long time to kill, how can we kill