
APACHE
Spark
TM

Persist and Caching in Spark

Grow Data Skills

The methods persist() and cache() in Apache Spark are used to save the RDD, DataFrame, or Dataset in memory for faster access during computation. They are effectively ways to optimize the execution of your Spark jobs, especially when you have repeated transformations on the same data. However, they differ in how they handle the storage:

1. **cache():** This method is a shorthand for using persist() with the default storage level. In other words, cache() is equivalent to calling persist() without any parameters. The default storage level is **MEMORY_AND_DISK** in **PySpark** and **MEMORY_AND_DISK_SER** in Scala Spark. This means that the RDD, DataFrame, or Dataset is stored in memory and, if it doesn't fit, the excess partitions are stored on disk.

2. **persist(storageLevel):** This method allows you to control how the data should be stored. You can pass a storage level as an argument to the persist() function, which gives you finer control over how the data is persisted. The storage level can be one of several options, each of which offers different trade-offs of memory usage and CPU efficiency, and can use either memory or disk storage or both.

Storage Levels in Persist

Grow Data Skills

- **StorageLevel.DISK_ONLY:** Store the RDD partitions only on disk.
- **StorageLevel.DISK_ONLY_2:** Same as the DISK_ONLY, but replicate each partition on two cluster nodes.
- **StorageLevel.MEMORY_AND_DISK:** Store RDD as deserialized objects in the JVM. If the RDD does not fit in memory, store the partitions that don't fit on disk, and read them from there when they're needed.
- **StorageLevel.MEMORY_AND_DISK_2:** Similar to MEMORY_AND_DISK, but replicate each partition on two cluster nodes.
- **StorageLevel.MEMORY_AND_DISK_SER:** Store RDD as serialized Java objects (one byte array per partition). This is generally more space-efficient than deserialized objects, especially when using a fast serializer, but more CPU-intensive to read.
- **StorageLevel.MEMORY_AND_DISK_SER_2:** Similar to MEMORY_AND_DISK_SER, but replicate each partition on two cluster nodes.
- **StorageLevel.MEMORY_ONLY**: Store RDD as deserialized Java objects in the JVM. If the RDD does not fit in memory, some partitions will not be cached and will be recomputed on the fly each time they're needed.
- **StorageLevel.MEMORY_ONLY_2:** Similar to MEMORY_ONLY, but replicate each partition on two cluster nodes.
- **StorageLevel.MEMORY_ONLY_SER:** Store RDD as serialized Java objects (one byte array per partition). This is more space-efficient, but more CPU-intensive to read.
- **StorageLevel.MEMORY_ONLY_SER_2:** Similar to MEMORY_ONLY_SER, but replicate each partition on two cluster nodes.

How does data skewness occur in Spark?

Grow Data Skills

Data skewness in Spark occurs when the data is not evenly distributed across partitions. This often happens when certain keys in your data have many more values than others. Consequently, tasks associated with these keys take much longer to run than others, which can lead to inefficient resource utilization and longer overall job execution time.

Here are a few scenarios where data skewness can occur:

- **Join Operations:** When you perform a join operation on two datasets based on a key, and some keys have significantly more values than others, these keys end up having larger partitions. The tasks processing these larger partitions will take longer to complete.

- **GroupBy Operations:** Similar to join operations, when you perform a groupByKey or reduceByKey operation, and some keys have many more values than others, data skewness can occur.

- **Data Distribution:** If the data distribution is not uniform, such that certain partitions get more data than others, then data skewness can occur. This could happen due to the nature of the data itself or the partitioning function not distributing the data evenly.

How to deal with data skewness ?

Grow Data Skills

Handling data skewness is a common challenge in distributed computing frameworks like Apache Spark. Here are some popular techniques to mitigate it:

- **Salting**: Salting involves adding a random component to a skewed key to create additional unique keys. After performing the operation (like a join), the extra key can be dropped to get back to the original data.
- **Splitting skewed data**: Identify the skewed keys and process them separately. For instance, you can filter out the skewed keys and perform a separate operation on them.
- **Increasing the number of partitions:** Increasing the number of partitions can distribute the data more evenly. However, this might increase the overhead of managing more partitions.
- **Using reduceByKey instead of groupByKey:** reduceByKey performs local aggregation before shuffling the data, which reduces the data transferred over the network.
- **Using Broadcast Variables:** When joining a large DataFrame with a small DataFrame, you can use broadcast variables to send a copy of the small DataFrame to all nodes. This avoids shuffling of the large DataFrame.

Difference between Repartition & Coalesce in Spark

Grow Data Skills

repartition():

- This method is used to increase or decrease the number of partitions in an RDD or DataFrame.
- It can cause a full shuffle of the data, i.e., all the data is reshuffled across the network to create new partitions.
- This operation is expensive due to the full shuffle. However, the resulting partitions can be balanced, i.e., they have roughly equal amounts of data.
- It can be used when you want to increase the number of partitions to allow for more concurrent tasks and increase parallelism when the cluster has more resources.
- In certain scenarios, you may want to partition based on a specific key to optimize your job. For example, if you frequently filter by a certain key, you might want all records with the same key to be on the same partition to minimize data shuffling. In such cases, you can use repartition() with a column name.

coalesce():

- This method is used to reduce the number of partitions in an RDD or DataFrame.
- It avoids a full shuffle. If you're decreasing the number of partitions, it will try to minimize the amount of data that's shuffled. Some partitions will be combined together, and the data within these partitions will not need to be moved.
- This operation is less expensive than repartition() because it minimizes data shuffling.
- However, it can lead to data skew if you have fewer partitions than before, because it combines existing partitions to reduce the total number.

Example for Salting

Grow Data Skills

Imagine we have two DataFrames, df1 and df2, that we want to join on a column named 'id'. Assume that the 'id' column is highly skewed.

Firstly, without any handling of skewness, the join might look something like this:

```
result = df1.join(df2, on='id', how='inner')
```

Now, let's implement salting to handle the skewness:

```
import pyspark.sql.functions as F

# Define the number of keys you'll use for salting
num_salting_keys = 100

# Add a new column to df1 for salting
df1 = df1.withColumn('salted_key', (F.rand()*num_salting_keys).cast('int'))

# Explode df2 into multiple rows by creating new rows with salted keys
df2_exploded = df2.crossJoin(F.spark.range(num_salting_keys).withColumnRenamed('id', 'salted_key'))

# Now perform the join using both 'id' and 'salted_key'
result = df1.join(df2_exploded, on=['id', 'salted_key'], how='inner')

# If you wish, you can drop the 'salted_key' column after the join
result = result.drop('salted_key')
```

In this code, we've added a "salt" to the 'id' column in df1 and created new rows in df2 for each salt value. We then perform the join operation on both 'id' and the salted key. This helps to distribute the computation for the skewed keys more evenly across the cluster.

Difference between RDD, Dataframe and Dataset

Grow Data Skills

Criteria	RDD (Resilient Distributed Dataset)	DataFrame	DataSet
Abstraction	Low level, provides a basic and simple abstraction.	High level, built on top of RDDs. Provides a structured and tabular view on data.	High level, built on top of DataFrames. Provides a structured and strongly-typed view on data.
Type Safety	Provides compile-time type safety, since it is based on objects.	Doesn't provide compile-time type safety, as it deals with semi-structured data.	Provides compile-time type safety, as it deals with structured data.
Optimization	Optimization needs to be manually done by the developer (like using mapreduce).	Makes use of Catalyst Optimizer for optimization of query plans, leading to efficient execution.	Makes use of Catalyst Optimizer for optimization.
Processing Speed	Slower, as operations are not optimized.	Faster than RDDs due to optimization by Catalyst Optimizer.	Similar to DataFrame, it's faster due to Catalyst Optimizer.
Ease of Use	Less easy to use due to the need of manual optimization.	Easier to use than RDDs due to high-level abstraction and SQL-like syntax.	Similar to DataFrame, it provides SQL-like syntax which makes it easier to use.
Interoperability	Easy to convert to and from other types like DataFrame and DataSet.	Easy to convert to and from other types like RDD and DataSet.	Easy to convert to and from other types like DataFrame and RDD.

Understand Spark-Submit

Grow Data Skills

spark-submit is a command-line interface to submit your Spark applications to run on a cluster. It can use a number of supported master URL's to distribute your application across a cluster, or can run the application locally.

Here is the general structure of the spark-submit command:

```
spark-submit \
  --class <main-class> \
  --master <master-url> \
  --deploy-mode <deploy-mode> \
  --conf <key>=<value> \
  <application-jar> \
  [application-arguments]
```

--class: This is the entry point for your application, i.e., where your main method runs. For Java and Scala, this would be a fully qualified class name.

--master: This is the master URL for the cluster. It can be a URL for any Spark-supported cluster manager. For example, local for local mode, spark://HOST:PORT for standalone mode, mesos://HOST:PORT for Mesos, or yarn for YARN.

Reference for list of configuration properties available in Spark 3.X - https://spark.apache.org/docs/latest/configuration.html

Understand Spark-Submit

Grow Data Skills

--deploy-mode: This can be either client (default) or cluster. In client mode, the driver runs on the machine from which the job is submitted. In cluster mode, the framework launches the driver inside the cluster.

--conf: This is used to set any Spark property. For example, you can set Spark properties like spark.executor.memory, spark.driver.memory, etc.

<application-jar>: This is a path to your compiled Spark application.

[application-arguments]: These are arguments that you need to pass to your Spark application.

For example, if you have a Spark job written in Python and you want to run it on a local machine, your spark-submit command might look like this:

```
spark-submit \
  --master local[4] \
  --py-files /path/to/other/python/files \
  /path/to/your/python/application.py \
  arg1 arg2
```

In this example, **--master local[4]** means the job will run locally with **4 worker threads (essentially, 4 cores)**. --py-files is used to add .py, .zip or .egg files to be distributed with your application. Finally, arg1 and arg2 are arguments that will be passed to your Spark application.

Understand Spark-Submit

Grow Data Skills

```
spark-submit \
  --master spark://10.0.0.1:7077 \
  --deploy-mode client \
  --executor-memory 4G \
  --driver-memory 2G \
  --conf spark.app.name=WordCountApp \
  --conf spark.executor.cores=2 \
  --conf spark.executor.instances=5 \
  --conf spark.default.parallelism=20 \
  --conf spark.driver.maxResultSize=1G \
  --conf spark.network.timeout=800 \
  --py-files /path/to/other/python/files.zip \
  /path/to/your/python/wordcount.py \
  /path/to/input/textfile.txt
```

In this example:

--master spark://10.0.0.1:7077 tells Spark to connect to a standalone Spark cluster at the given IP address.

--deploy-mode client tells Spark to run the driver program on the machine that the job is submitted from, as opposed to within one of the worker nodes.

Understand Spark-Submit

Grow Data Skills

--executor-memory 4G and **--driver-memory 2G** set the amount of memory for the executor and driver processes, respectively.

--**conf spark.app.name=WordCountApp** sets a name for your application, which can be useful when monitoring your cluster.

--conf spark.executor.cores=2 sets the number of cores to use on each executor.

--conf spark.executor.instances=5 sets the number of executor instances

--conf spark.default.parallelism=20 sets the default number of partitions in RDDs returned by transformations like join(), reduceByKey(), and parallelize() when not set by user.

--conf spark.driver.maxResultSize=1G limits the total size of the serialized results of all partitions for each Spark action (e.g., collect). This should be at least as large as the largest object you want to collect.

--conf spark.network.timeout=800 sets the default network timeout value to 800 seconds. This configuration plays a vital role in cases where you deal with large shuffles.

--py-files /path/to/other/python/files.zip adds Python .zip, .egg or .py files to distribute with your application. If you have Python files that your wordcount.py depends on, you can bundle them into a .zip or .egg file and pass it with --py-files.

/path/to/your/python/wordcount.py is the path to the Python file that contains your Spark application.

/path/to/input/textfile.txt is an argument that will be passed to your Spark application. In this case, it's a path to the text file that your WordCount application will process.

Memory Management in Spark

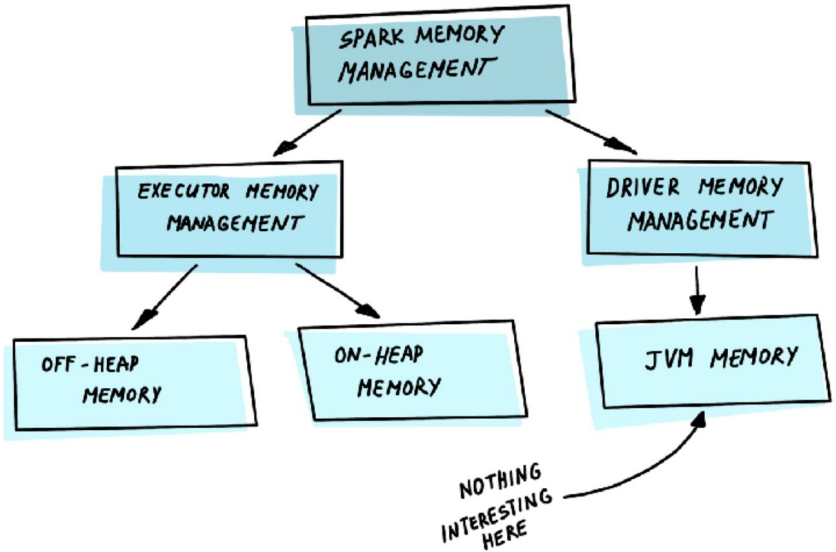
Grow Data Skills

When the Spark application is launched, the Spark cluster will start two processes — **Driver** and **Executor**.

The driver is a master process responsible for creating the Spark context, submission of Spark jobs, and translation of the whole Spark pipeline into computational units — tasks. It also coordinates task scheduling and orchestration on each Executor.

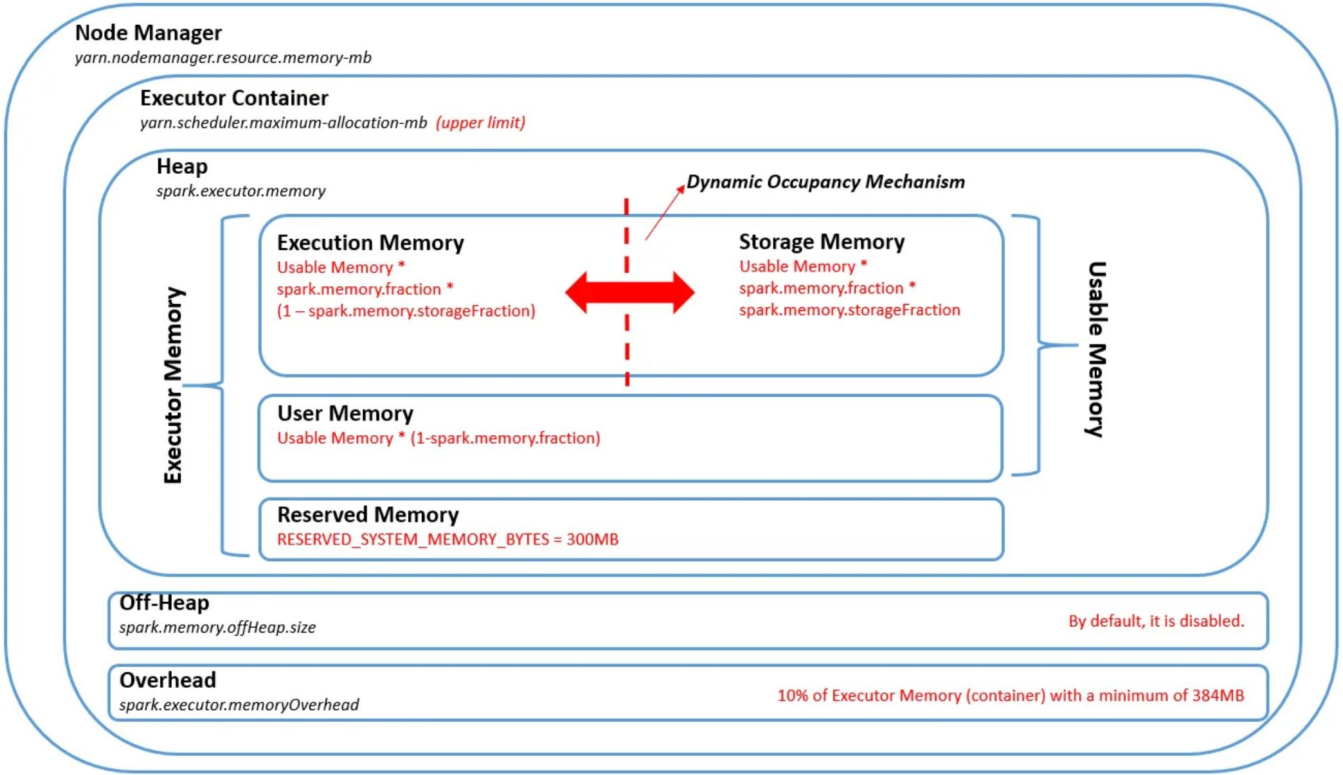
Driver memory management is not much different from the typical JVM process and therefore will not be discussed further.

The executor is responsible for **performing specific computational tasks** on the worker nodes and returning the results to the driver, as well as providing storage for RDDs. And **its internal memory management is very interesting.**



Memory Management - Executor Container

Grow Data Skills



When submitting a Spark job in a cluster with Yarn, Yarn allocates Executor containers to perform the job on different nodes.

ResourceManager handles memory requests and allocates executor container up to maximum allocation size settled by **yarn.scheduler.maximum-allocation-mb** configuration. Memory requests higher than the specified value will not take effect.

Memory Management - Executor Container

Grow Data Skills

On a single node, it is done by **NodeManager**. NodeManager has an **upper limit** of resources available to it because it is limited by the resources of one node of the cluster. In Yarn it set up by **yarn.nodemanager.resource.memory-mb** configuration. It is the amount of physical memory per NodeManager, in MB, which can be allocated for yarn containers.

One **ExecutorContainer** is just one JVM. And the entire **ExecutorContainer** memory area is divided into three sections:

Heap Memory: This is the JVM heap memory where Spark runs its operations and stores data. It's further divided into:

- **Execution Memory:** Used for computation in shuffles, joins, sorts, and aggregations.
- **Storage Memory:** Used for caching and propagating internal data across the cluster.

The size of the executor heap memory is controlled by the spark.executor.memory configuration property.

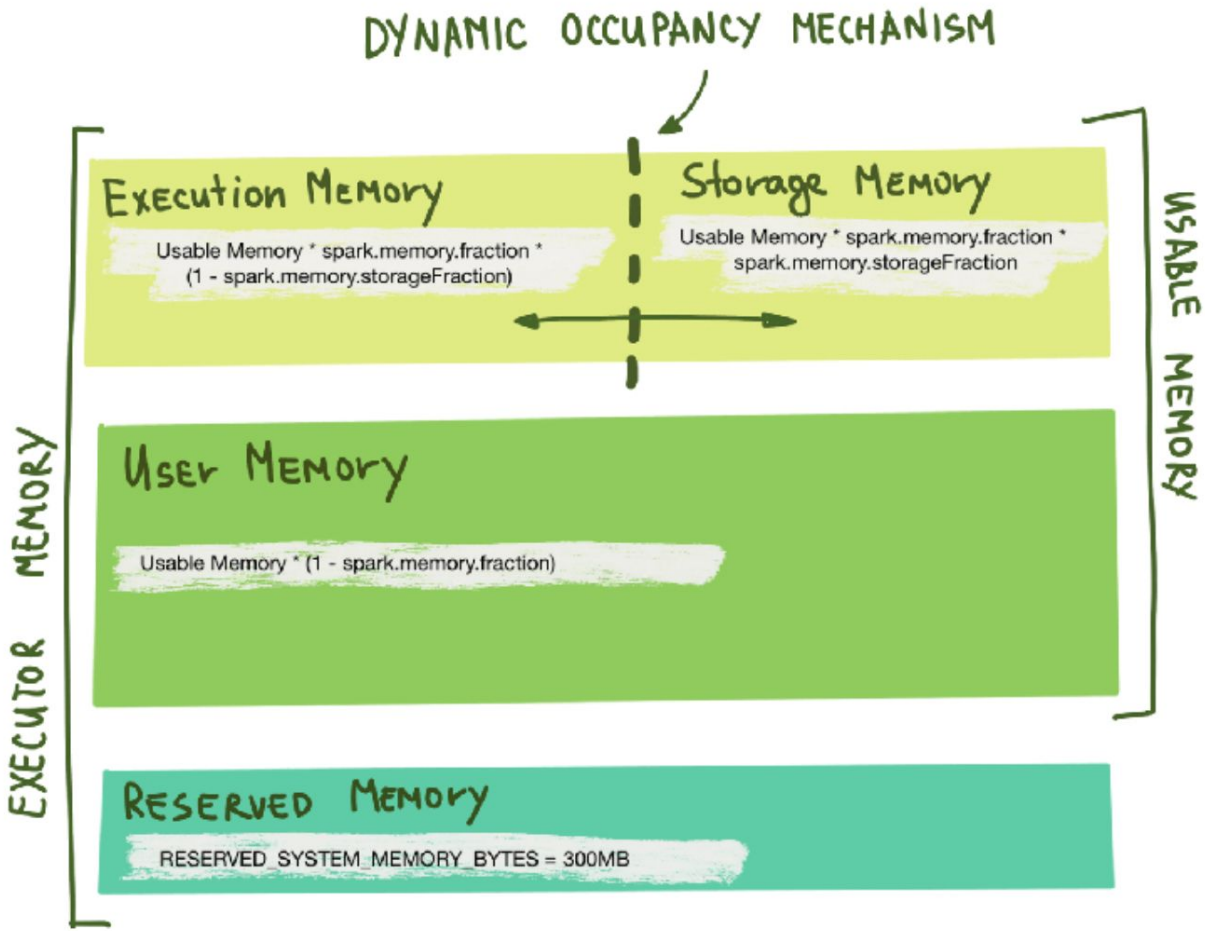
Off-Heap Memory: This is the memory managed directly by the application, not by the JVM. This memory is not subject to Java's Garbage Collector. Spark's core transformations and actions do not use off-heap memory by default. This can be configured using **spark.executor.memoryOffHeap.enabled** and **spark.executor.memoryOffHeap.size**

Overhead Memory (Non-Heap Memory): This is memory used by Spark for things outside of the executor heap memory. It includes JVM overheads, interned strings, other native overheads, etc. In containerized environments (like YARN or Kubernetes), it also accounts for non-JVM memory consumption (like Python processes).

The size of overhead memory is by default **10% of the executor memory** with a **minimum** of **384 MB**. This can be configured manually using **spark.executor.memoryOverhead** (absolute size) or **spark.executor.memoryOverheadFactor** (fraction of executor memory).

Memory Management - Executor Container

Grow Data Skills



Memory Management - Executor Container

Grow Data Skills

Reserved Memory: The most boring part of the memory. Spark reserves this memory to store internal objects. It guarantees to reserve sufficient memory for the system even for small JVM heaps.

Reserved Memory is hardcoded and equal to **300 MB** (value RESERVED_SYSTEM_MEMORY_BYTES in source code). In the test environment (when spark.testing set) we can modify it with spark.testing.reservedMemory.

Storage Memory: Storage Memory is used for caching and broadcasting data. Storage Memory size can be found by:

Storage Memory = usableMemory * spark.memory.fraction * spark.memory.storageFraction

Storage Memory is 30% of all system memory by default (1 * 0.6 * 0.5 = 0.3).

Execution Memory: It is mainly used to store temporary data in the shuffle, join, sort, aggregation, etc. Most likely, if your pipeline runs too long, the problem lies in the lack of space here.

Execution Memory = usableMemory * spark.memory.fraction * (1 - spark.memory.storageFraction)

As Storage Memory, Execution Memory is also equal to 30% of all system memory by default (1 * 0.6 * (1 - 0.5) = 0.3).

Memory Management - Executor Container

Grow Data Skills

User Memory: It is mainly used to store data needed for RDD conversion operations, such as lineage. You can store your own data structures there that will be used inside transformations. It's up to you what would be stored in this memory and how. Spark makes completely no accounting on what you do there and whether you respect this boundary or not.

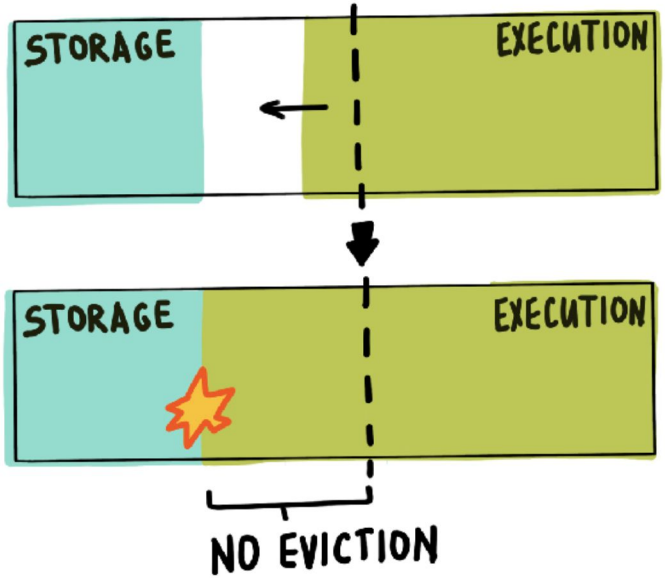
User Memory = usableMemory * (1 - spark.memory.fraction)

It is 1 * (1 - 0.6) = 0.4 or 40% of available memory by default.

Dynamic Occupancy Mechanism

Grow Data Skills

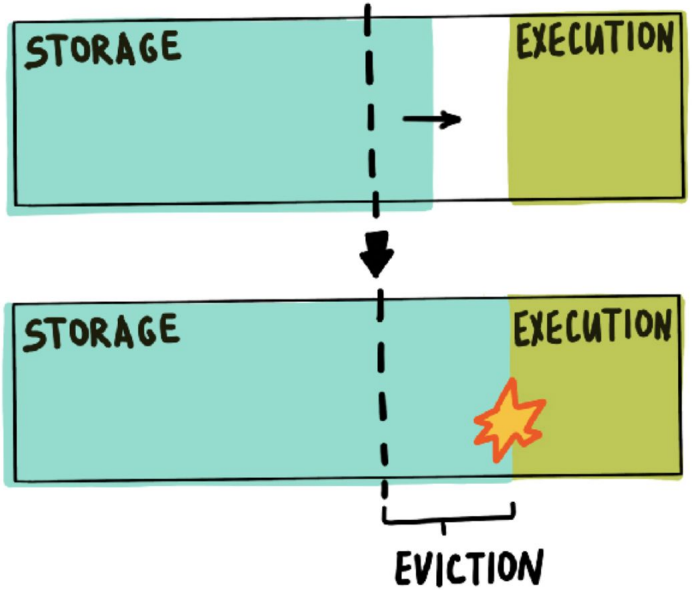
Execution and Storage have a shared memory. They can borrow it from each other. This process is called the Dynamic occupancy mechanism.



There are two parts of the shared memory — the Storage side and the Execution side. The shared Storage memory can be used up to a certain threshold. In the code, this threshold is called onHeapStorageRegionSize. This part of memory is used by Storage memory, but only if it is not occupied by Execution memory. Storage memory has to wait for the used memory to be released by the executor processes. The default size of onHeapStorageRegionSize is all Storage Memory.

Dynamic Occupancy Mechanism

Grow Data Skills



When Execution memory is not used, Storage can borrow as much Execution memory as available until execution reclaims its space. When this happens, cached blocks will be evicted from memory until sufficient borrowed memory is released to satisfy the Execution memory request.

The creators of this mechanism decided that Execution memory has priority over Storage memory. They had reasons to do so — the execution of the task is more important than the cached data, the whole job can crash if there is an OOM in the execution.

How to process 1 TB of data in Apache Spark ?

Grow Data Skills

Total blocks - 128MB default block size (HDFS) - 1TB -> 1 *1024 1024/128 = 8192.

Let us take
-> 20 Executor-machine
-> 5 Core processor in each executor node
-> 6 GB RAM in each executor node

And the cluster can perform (20*5=100) Task parallel at a time, Here tasks mean block so 100 blocks can be processed parallelly at a time.

100*128MB = 12800 MB / 1024GB = 12.5 GB (So, 12GB data will get processed in 1st set of a batch)

Since the RAM size is 6GB in each executor, (20 executor x 6GB RAM =120GB Total RAM) So, at a time 12GB of RAM will occupy in a cluster (20node/12gb=1.6GB RAM In each executor).

Now, Available RAM in each executor will be (6GB - 1.6GB = 4.4GB) RAM which will be reserved for other users' jobs and programs.

So, 1TB = 1024 GB / 12GB = (Whole data will get processed in around 85 batches).

Note :- Actual values may differ in comparison with real-time scenarios.

Case studies for resource allocation

Grow Data Skills

Case 1 Hardware — 6 Nodes and each node have 16 cores, 64 GB RAM

We start with how to choose the number of cores:

Number of cores = Concurrent tasks an executor can run

So we might think, more concurrent tasks for each executor will give better performance. But research shows that any application with more than 5 concurrent tasks, would lead to a bad show. **So the optimal value is 5.**

This number comes from the ability of an executor to run parallel tasks and not from how many cores a system has. So the number 5 stays the same even if we have double (32) cores in the CPU

- 5 cores per executor
  - For max HDFS throughput
- Cluster has 6 * 15 = 90 cores in total (after taking out Hadoop/Yarn daemon cores)
- 90 cores / 5 cores/executor = 18 executors
- 1 executor for AM => 17 executors
- Each node has 3 executors
- 63 GB/3 = 21 GB, 21 x (1-0.07) ~ 19 GB (counting off heap overhead)

Case studies for resource allocation

Grow Data Skills

Number of executors:

Coming to the next step, with 5 as cores per executor, and 15 as total available cores in one node (CPU) — we come to 3 executors per node which is 15/5. We need to calculate the number of executors on each node and then get the total number for the job.

So with 6 nodes and 3 executors per node — we get a total of 18 executors. Out of 18, we need 1 executor (java process) for Application Master in YARN. So the final number is 17 executors

This 17 is the number we give to spark using –num-executors while running from the spark-submit shell command.

Memory for each executor:

From the above step, we have 3 executors per node. And available RAM on each node is 63 GB

So memory for each executor in each node is 63/3 = 21GB.

However small overhead memory is also needed to determine the full memory request to YARN for each executor.

The formula for that overhead is max(384, .07 * spark.executor.memory) , Calculating that overhead: .07 * 21 (Here 21 is calculated as above 63/3) = 1.47

Since 1.47 GB > 384 MB, the overhead is 1.47

Take the above from each 21 above => 21–1.47 ~ 19 GB , So executor memory — 19 GB

Final numbers — Executors — 17, Cores 5, Executor Memory — 19 GB

Case studies for resource allocation

Case 2 Hardware — 6 Nodes and Each node have 32 Cores, 64 GB

Number of cores of 5 is the same for good concurrency as explained above.

Number of executors for each node = 32/5 ~ 6

So total executors = 6 * 6 Nodes = 36. Then the final number is 36–1(for AM) = 35

Executor memory:

6 executors for each node. 63/6 ~ 10. Overhead is .07 * 10 = 700 MB. So rounding to 1GB as overhead, we get 10–1 = 9 GB

Final numbers — Executors — 35, Cores 5, Executor Memory — 9 GB

Case 3 — When more memory is not required for the executors

The above scenarios start with accepting the number of cores as fixed and moving to the number of executors and memory.

Now for the first case, if we think we do not need 19 GB, and just 10 GB is sufficient based on the data size and computations involved, then following are the numbers:

Cores: 5

Number of executors for each node = 3. Still, 15/5 as calculated above.

At this stage, this would lead to 21 GB, and then 19 as per our first calculation. But since we thought 10 is ok (assume little overhead), then we cannot switch the number of executors per node to 6 (like 63/10). Because with 6 executors per node and 5 cores it comes down to 30 cores per node when we only have 16 cores. So we also need to change the number of cores for each executor.

So calculating again,

The magic number 5 comes to 3 (any number less than or equal to 5). So with 3 cores, and 15 available cores — we get 5 executors per node, 29 executors (which is (5*6 -1)) and memory is 63/5 ~ 12.

The overhead is 12*.07=.84. So executor memory is 12–1 GB = 11 GB

Final Numbers are 29 executors, 3 cores, executor memory is 11 GB

Broadcast and Accumulators in Spark

Grow Data Skills

Broadcast Variables:

In Spark, broadcast variables are read-only shared variables that are cached on each worker node rather than sent over the network with tasks. They're used to give every node a copy of a large input dataset in an efficient manner. Spark's actions are executed through a set of stages, separated by distributed "shuffle" operations. Spark automatically broadcasts the common data needed by tasks within each stage.

```
from pyspark.sql import SparkSession

# initialize SparkSession
spark = SparkSession.builder.getOrCreate()

# broadcast a large read-only lookup table
large_lookup_table = {"apple": "fruit", "broccoli": "vegetable", "chicken": "meat"}
broadcasted_table = spark.sparkContext.broadcast(large_lookup_table)

def classify(word):
    # access the value of the broadcast variable
    lookup_table = broadcasted_table.value
    return lookup_table.get(word, "unknown")

data = ["apple", "banana", "chicken", "potato", "broccoli"]
rdd = spark.sparkContext.parallelize(data)
classification = rdd.map(classify)

print(classification.collect())
```

Broadcast and Accumulators in Spark

Grow Data Skills

Accumulators:

Accumulators are variables that are only "added" to through associative and commutative operations and are used to implement counters and sums efficiently. They can be used to implement counters (as in MapReduce) or sums. Spark natively supports accumulators of numeric types, and programmers can add support for new types.

A simple use of accumulators is:

```
from pyspark.sql import SparkSession

# initialize SparkSession
spark = SparkSession.builder.getOrCreate()

# create an Accumulator[Int] initialized to 0
accum = spark.sparkContext.accumulator(0)

rdd = spark.sparkContext.parallelize([1, 2, 3, 4])
def add_to_accum(x):
    global accum
    accum += x

rdd.foreach(add_to_accum)

# get the current value of the accumulator
print(accum.value)
```

Handling Failures in Spark

Grow Data Skills

Driver Failure:

The driver program runs the main() function of the application and creates a SparkContext. If the driver node fails, the entire application will be terminated, as it's the driver program that declares transformations and actions on data and submits such requests to the cluster.

Impact:

- The driver node is a single point of failure for a Spark application.
- If the driver program fails due to an exception in user code, the entire Spark application is terminated, and all executors are released.

Handling Driver Failure:

- Driver failure is usually fatal, causing the termination of the application.
- It's crucial to handle exceptions in your driver program to prevent such failures.
- Also, monitor the health of the machine hosting the driver program to prevent failures due to machine errors.
- In some cluster managers like Kubernetes, Spark supports mode like spark.driver.supervise to supervise and restart the driver on failure.

Handling Failures in Spark

Grow Data Skills

Executor Failure:

Executors in Spark are responsible for executing the tasks. When an executor fails, the tasks that were running will fail.

Impact:

- Executors can fail for various reasons, such as machine errors or OOM errors in the user's application.
- If an executor fails, the tasks that were running on it are lost.
- The failure of an executor doesn't cause the failure of the Spark application, unless all executors fail.

Handling Executor Failure:

- If an executor fails, Spark can reschedule the failed tasks on other executors.
- There is a certain threshold for task failures. If the same task fails more than 4 times (default), the application will be terminated.
- Make sure to tune the resources allocated for each executor, as an executor might fail due to insufficient resources.
- For resilience, you can also opt to replicate the data across different executor nodes.

Failure Due to Out Of Memory in Spark

Grow Data Skills

Spark Driver OOM Scenarios:

- **Large Collect Operations:** If the data collected from executors using actions such as collect() or take() is too large to fit into the driver's memory, an OutOfMemoryError will occur.
    - **Solution**: Be cautious with actions that pull large volumes of data into the driver program. Use actions like take(n), first(), collect() carefully, and only when the returned data is manageable by the driver.
- **Large Broadcast Variables**: If a broadcast variable is larger than the amount of free memory on the driver node, this will also cause an OOM error.
    - **Solution**: Avoid broadcasting large variables. If possible, consider broadcasting a common subset of the data, or use Spark's built-in broadcast join if joining with a large DataFrame.
- **Improper Driver Memory Configuration:** If spark.driver.memory is set to a high value, it can cause the driver to request more memory than what is available, leading to an OOM error.
    - **Solution**: Set the spark.driver.memory config based on your application's need and ensure it doesn't exceed the physical memory limits.

Failure Due to Out Of Memory in Spark

Grow Data Skills

Spark Executor OOM Scenarios:

- **Large Task Results:** If the result of a single task is larger than the amount of free memory on the executor node, an OutOfMemoryError will occur.
    - Solution: Avoid generating large task results. This is often due to a large map operation. Consider using reduceByKey or aggregateByKey instead of groupByKey when transforming data.
- **Large RDD or DataFrame operations**: Certain operations on RDDs or DataFrames, like join, groupByKey, reduceByKey, can cause data to be shuffled around, leading to a large amount of data being held in memory at once, potentially causing an OOM error.
    - Solution: Be cautious with operations that require shuffling large amounts of data. Use operations that reduce the volume of shuffled data, such as reduceByKey and aggregateByKey, instead of groupByKey.
- **Persistent RDDs/DataFrames:** If you're persisting many RDDs/DataFrames in memory and there isn't enough memory to store them, this will also cause an OOM error.
    - Solution: Unpersist unnecessary RDDs and DataFrames as soon as they are no longer needed. Tune the spark.memory.storageFraction to increase the amount of memory reserved for cached RDDs/DataFrames.
- **Improper Executor Memory Configuration:** Similar to the driver, if spark.executor.memory is set to a high value, it can cause the executor to request more memory than what is available, leading to an OOM error.
    - Solution: Set the spark.executor.memory config based on your application's need and ensure it doesn't exceed the physical memory limits of the executor nodes.

Code Level and Resource Level Optimizations in Spark

Grow Data Skills

Code Level Optimization:

- ***Use DataFrames/Datasets instead of RDDs:*** DataFrames and Datasets have optimized execution plans, leading to faster and more memory-efficient operations than RDDs. They also have more intuitive APIs for many operations.
- ***Leverage Broadcasting:*** If you're performing an operation like a join between a large DataFrame and a small DataFrame, consider broadcasting the smaller DataFrame. Broadcasting sends the smaller DataFrame to all worker nodes, so they have a local copy and don't need to fetch the data across the network.
- ***Avoid Shuffling:*** Operations like groupByKey cause shuffling, where data is transferred across the network, which can be slow. Operations like reduceByKey or aggregateByKey reduce the amount of data that needs to be shuffled, and can be faster.
- ***Avoid Collecting Large Data:*** Be careful with operations like collect() that bring a large amount of data into the driver program, which could cause an out of memory error.
- ***Repartitioning and Coalescing:*** Depending on your use case, you might want to increase or decrease the number of partitions. If you have too many small partitions, use coalesce to combine them. If you have too few large partitions, use repartition to split them.
- ***Persist/Cache Wisely:*** Persist or cache the DataFrames or RDDs that you'll reuse. However, keep in mind that these operations consume memory, so use them judiciously.

Code Level and Resource Level Optimizations in Spark

Grow Data Skills

Resource Configuration Optimization:

- ***Tune Memory Parameters:*** Make sure to set spark.driver.memory, spark.executor.memory, spark.memory.fraction, and spark.memory.storageFraction based on the memory requirements of your application and the capacity of your hardware.

- ***Control Parallelism:*** Use spark.default.parallelism and spark.sql.shuffle.partitions to control the number of tasks during operations like join, reduceByKey, etc. Too many tasks can cause a lot of overhead, but too few tasks might not fully utilize your cluster.

- ***Dynamic Allocation:*** If your cluster manager supports it, use dynamic resource allocation, which allows Spark to dynamically adjust the resources your application occupies based on the workload. This means that if your application has stages that require lots of resources, they can be allocated dynamically.

  spark.dynamicAllocation.enabled true
  spark.dynamicAllocation.initialExecutors 2
  spark.dynamicAllocation.minExecutors 1
  spark.dynamicAllocation.maxExecutors 20
  spark.dynamicAllocation.schedulerBacklogTimeout 1m
  spark.dynamicAllocation.sustainedSchedulerBacklogTimeout 2m
  spark.dynamicAllocation.executorIdleTimeout 2min

  - ***spark.dynamicAllocation.enabled*** is set to true to enable dynamic allocation.
  - ***spark.dynamicAllocation.initialExecutors*** is set to 2 to specify that initially, two executors will be allocated.
  - ***spark.dynamicAllocation.minExecutors*** and ***spark.dynamicAllocation.maxExecutors*** control the minimum and maximum number of executors, respectively.
  - ***spark.dynamicAllocation.schedulerBacklogTimeout*** and ***spark.dynamicAllocation.sustainedSchedulerBacklogTimeout*** control how long a backlog of tasks Spark will tolerate before adding more executors.
  - ***spark.dynamicAllocation.executorIdleTimeout*** controls how long an executor can be idle before Spark removes it.

Code Level and Resource Level Optimizations in Spark

Resource Configuration Optimization:

- ***Tune Garbage Collection:*** Spark uses the JVM, so the garbage collector can significantly affect performance. You can use spark.executor.extraJavaOptions to pass options to the JVM to tune the garbage collection.

- ***Use Appropriate Data Structures:*** Parquet and Avro are both columnar data formats that are great for analytical queries and schema evolution. If your data processing patterns match these, consider using these formats.

Best practices to design Spark application

Grow Data Skills

Understand Your Data and Workload:

- **Data Size:** Understand the volume of data your application will process. This will influence the infrastructure needed and the parallelism level in your application.
- **Data Skewness:** Identify if your data is skewed, as this can cause certain tasks to take longer to complete and negatively impact performance. Techniques like key salting can be applied to handle skewness.
- **Workload Type:** Understand the type of operations your application will perform. For example, analytical workloads may benefit from columnar data formats like Parquet or ORC.

Application Design:

- **Transformations**: Use transformations like map, filter, reduceByKey over actions as much as possible, as transformations are lazily evaluated and can benefit from Spark's optimization.
- **Shuffling**: Minimize operations that cause data shuffling across the network, as they are expensive. Avoid operations like groupByKey in favor of reduceByKey or aggregateByKey.
- **Broadcasting**: Small datasets that are used across transformations should be broadcasted to improve performance.
- **Partitioning**: Use the right level of partitioning. Too few partitions can lead to fewer concurrent tasks and underutilization of resources. Too many partitions might lead to excessive overhead.

Best practices to design Spark application

Grow Data Skills

Resource Allocation and Configuration:

- **Memory Allocation**: Properly allocate memory to Spark executors, driver, and overhead. Monitor the memory usage of your application to fine-tune these settings.
- **Dynamic Resource Allocation:** Enable dynamic resource allocation if supported by your cluster manager. This allows Spark to adjust the resources based on workload.
- **Parallelism**: Configure the level of parallelism (number of partitions) based on the data volume and infrastructure.

Infrastructure Consideration:

- **Storage**: Use fast storage (like SSDs) to store your data, as I/O operations can become a bottleneck in large data processing.
- **Network**: A high-speed network is important, especially if your workload involves data shuffling.
- **Nodes and Cores:** More nodes with multiple cores can increase parallelism and data processing speed.
- **Data Locality**: Aim for data locality, i.e., running tasks on nodes where the data is stored, to reduce network I/O.

Best practices to design Spark application

Grow Data Skills

Monitor and Iterate:

- Use Spark's built-in web UI to monitor your applications. Look for stages that take a long time to complete, tasks that fail and are retried, storage and computation bottlenecks, and executor memory usage.