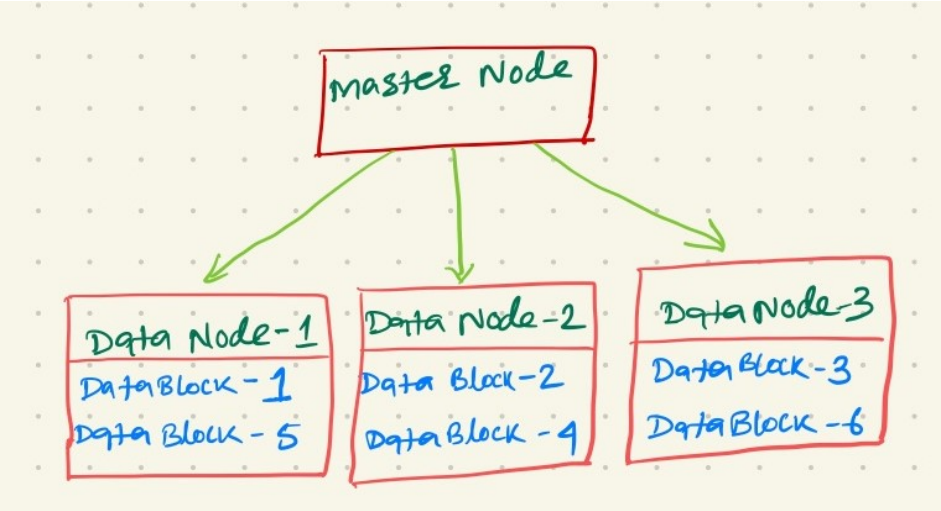
Chapter: One
Spark Internals

In a distributed storage, like HDFS, data files would be distributed across multiple data nodes. A simple 4 nodes cluster like below, usually have a master node and 3 data nodes.



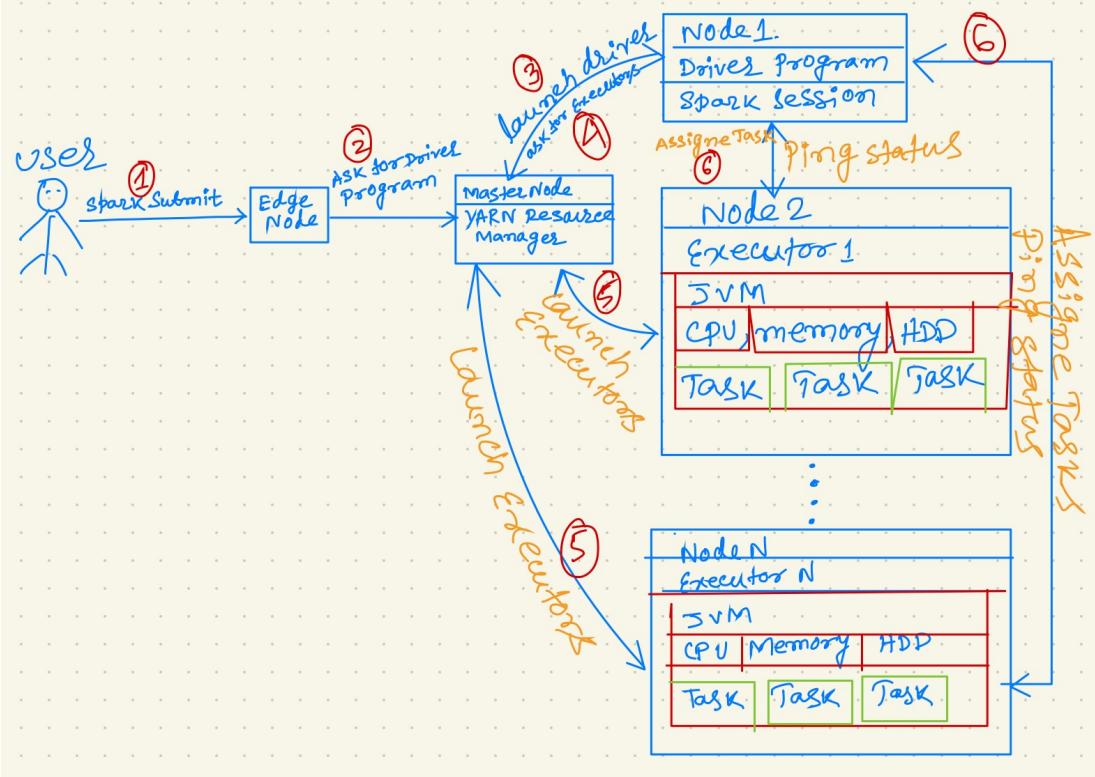
Master node and data nodes

Once we submit the spark program below are the steps, it goes through to execute the program.

1. Client request to cluster manager for a spark driver process.
2. Once request is granted, cluster manager then launch a driver program onto a node. It can be any available node in the cluster (cluster mode) or the node we are connected(client mode), based on the deploy mode we had specified.
3. Once spark has the driver program running, it will start a spark session and begin executing the user program.
4. The spark session will communicate with the cluster manager and

request to launch spark executor process across the cluster. The number of executors are depended on the user configuration. If no configuration is specified then spark will pick up the configuration form the default config file.

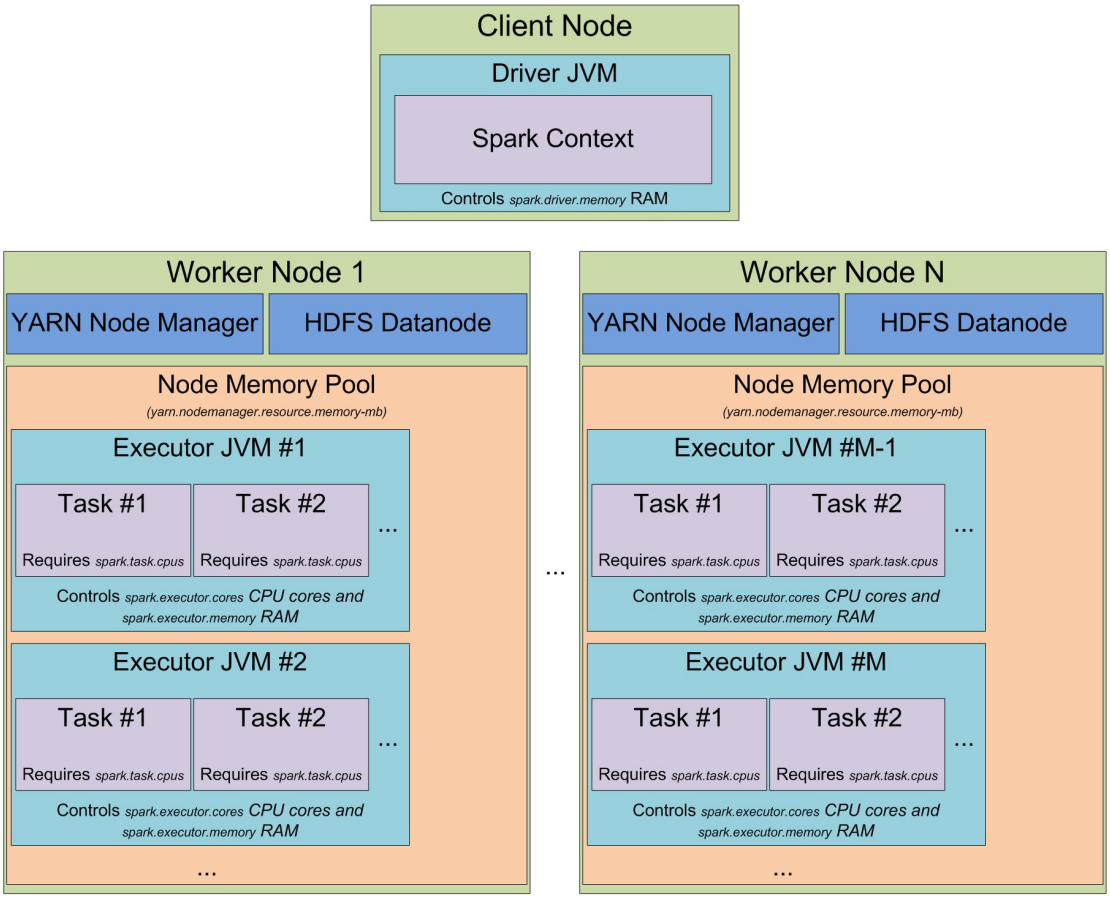
5. Then cluster manager will launch all the spark executors and send their locations to the driver process.
6. Spark driver and executors communicate between themselves and start executing the code. Spark driver is responsible for managing and executing the program.



How spark cluster gets launched

7. Spark driver read the code and convert that code to DAG of RDDs to represent the computation.
8. Then the DAG gets converted to to logical plans, then to a suitable physical plan.
9. Spark driver then converts the physical plan to Jobs. One job can have multiple stages and one stage can have multiple tasks. Task is the unit of execution. One task run on one cpu and one partition of

the data. This is the reason splittable format is important, so spark can run multiple tasks in parallel. Based on the resources spark can run multiple executors in one worker nodes and one executor can have multiple tasks, as shown in below picture.



Multiple executors and tasks in worker nodes

10. Spark driver schedules the tasks onto each worker, and each worker responds to the status of the task (success or failure) to driver.
11. If required, spark driver can request to cluster manager for more executors. Vice verse is also true, if not all executors are required cluster manager can shut down the executors. We can define all of these properties in spark's dynamic resource allocation.
12. Once application completes, the driver sends the status to cluster manager and exists.
13. Then cluster manager shuts down the executors in that spark cluster

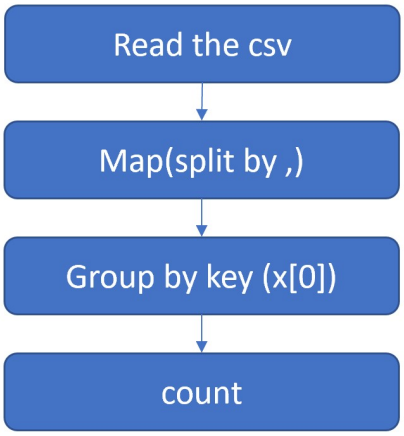
for the driver.

Example: Very simple example. Read a csv file then group the data by first column and count the values.

```
# python code
sc.textFile('/mnt/e/Training_Data/5m-Sales-Records/*.csv') \
.map(lambda x: x.split(',')) \
.groupBy(lambda x: x[0]) \
.count()
```

Step 1: Create the RDD (DAG)

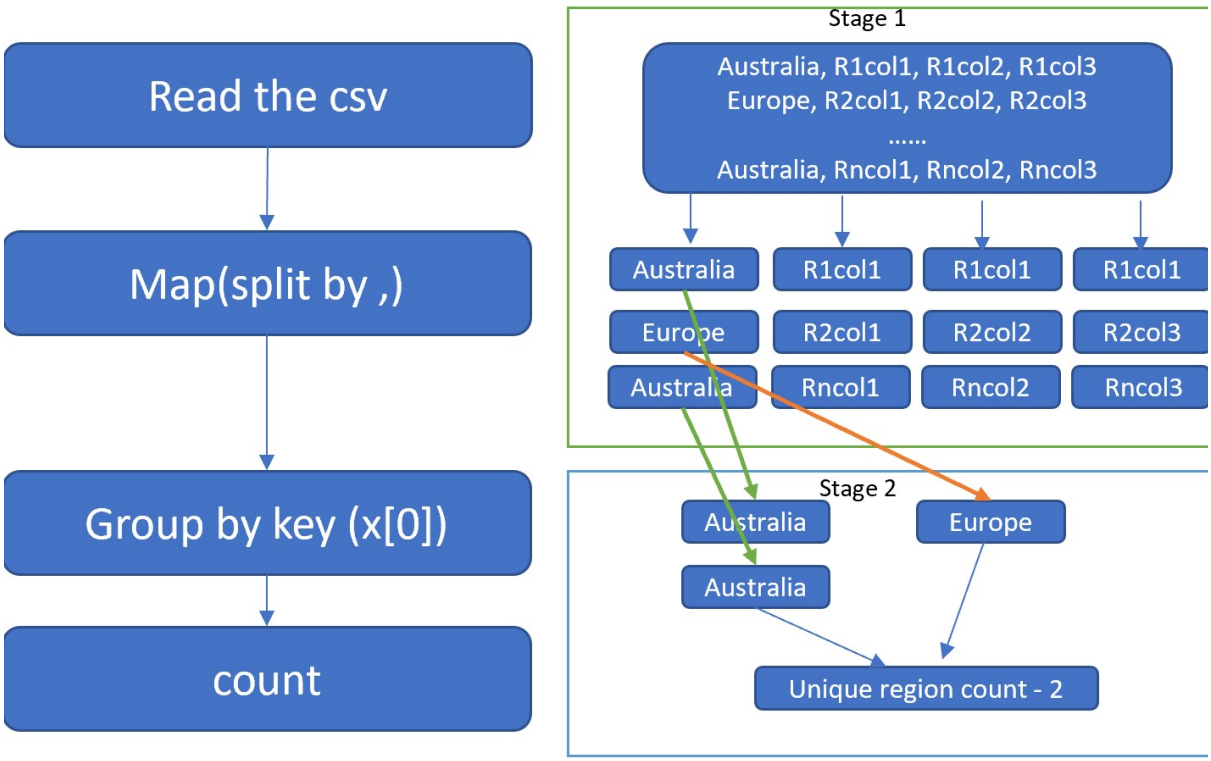
This is the logical protestation of the job.



Logical break up

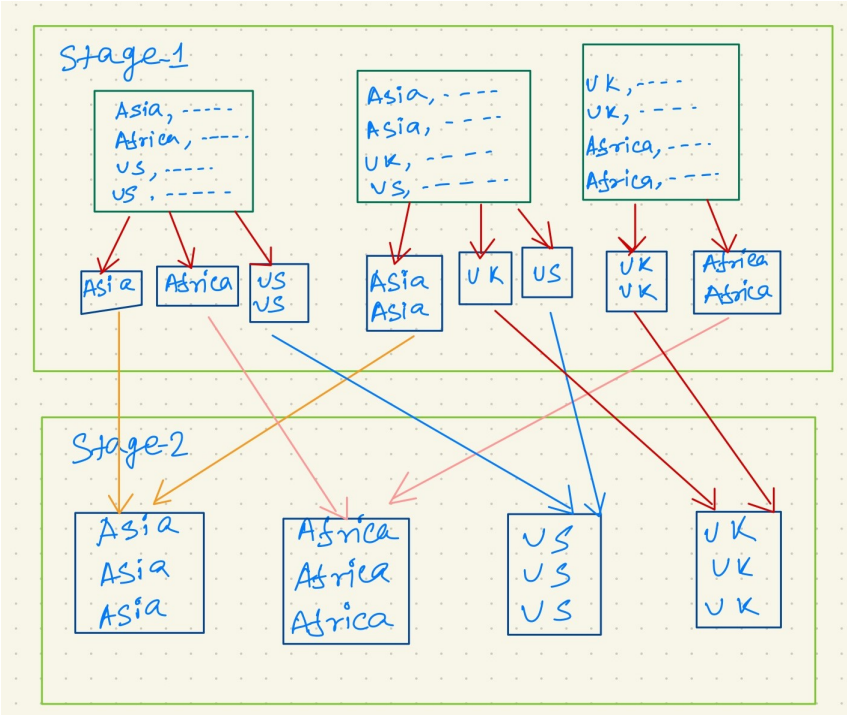
Step 2: Create execution plan

1. Fuse or pipeline process as mush as possible so spark executor don't have to go through the data multiple times. In our example
2. Create a new stage whenever data need to be shuffle or reorganize.



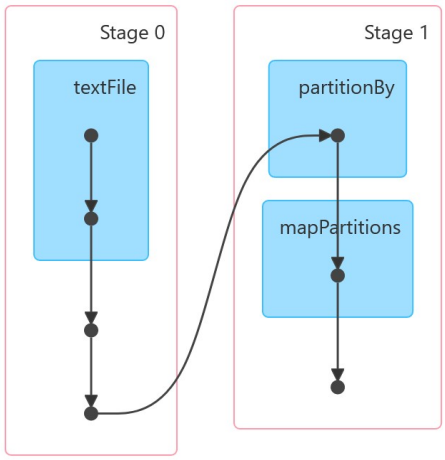
Different stages

- This process will create a job.
- Reading csv and splitting the data by delimiter can be done with out reorganize the data.
- To group the data by region and count it, needs data to shuffle and put all the same region in single partition.
- During shuffle first stage writes the data and then same data gets read by next stage. In the bellow picture you can notice the shuffle write and shuffle read.
- Shuffle is pull based, not pushed based. It means that next stage will pull the data from previous stage. Due to memory limit Shuffle data can spill and spark will write the intermediate data into disk. This is called Shuffle Spill. This is one reason that shuffle is slower.



Shuffle between stages

- So the job is splitted into two stages.

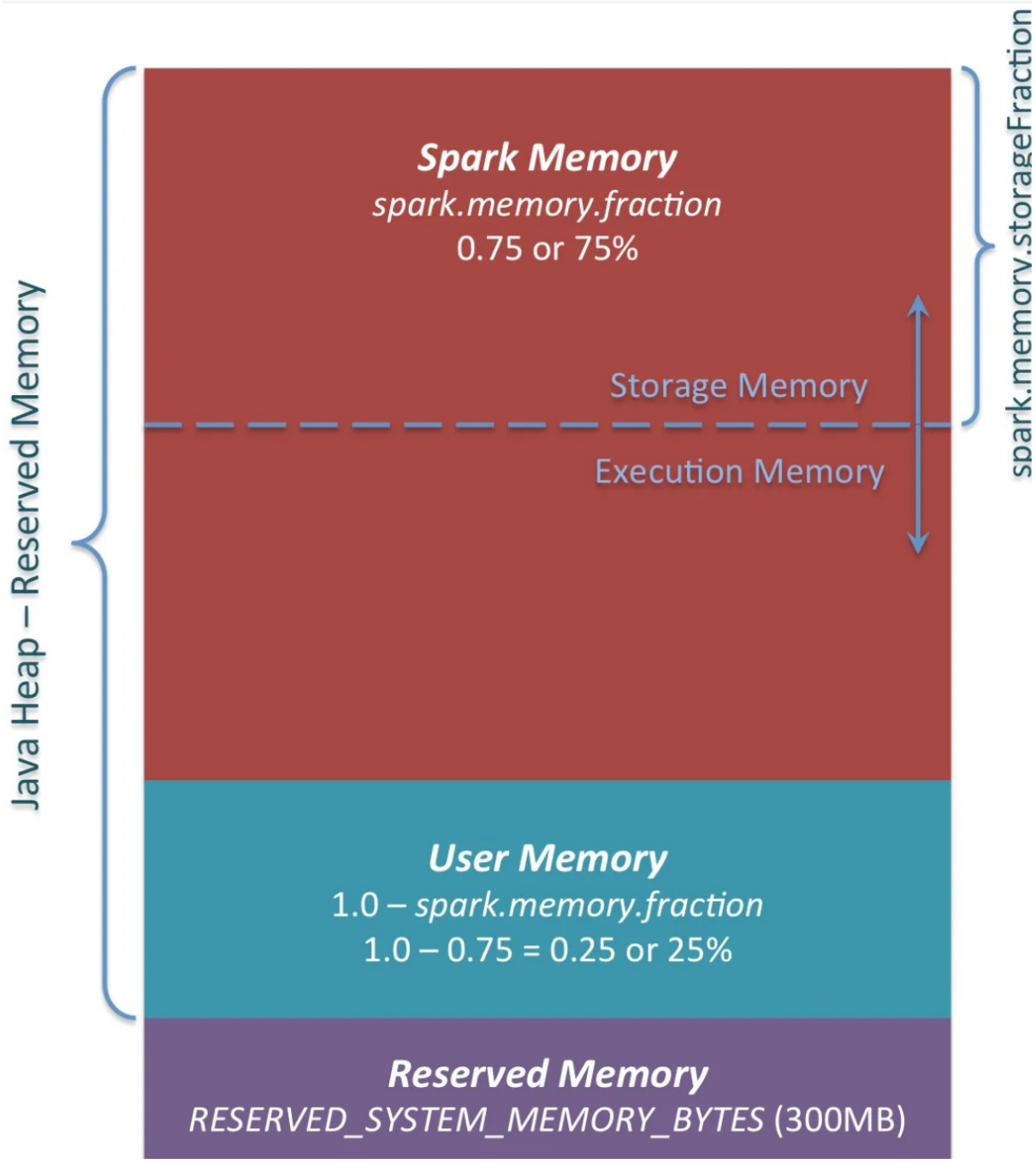


Two stages

	Description		Submitted	Duration	Tasks: Succeeded/Total	Input	Output	Shuffle Read	Shuffle W
	count at <stdin>:1	+details	2021/05/23 08:11:56	36 s	20/20			376.4 MiB	
	groupBy at <stdin>:1	+details	2021/05/23 08:11:32	24 s	20/20	624.4 MiB			376.4 MiB

shuffle read and write of different stages

Spark Memory Management: From v1.6+ onwards spark has been using unified memory management. Below is the capture of how unified memory management looks like.



spark Unified Memory Manager

Reserved Memory: This is a fixed memory reserved for spark internal objects. This is not user accessible and it's fixed as 300MB.

Note: While calculating the memory usage you must consider reserver memory as well, if the executor memory is less than *1.5* times of reserved memory, Spark will fail with a *"please use larger heap size"* error message.

User Memory: This is the memory completely used by user and not managed by spark. You can use this for defining the custom data structure, like hash table for map partition. User also responsible for out of memory issues in user memory. This space is controlled by *spark.memory.fraction(default = 0.75)* parameter.

Example: For 4GB heap user memory = (Java Heap — Reserved Memory) * (1.0 — spark.memory.fraction) which is (4024MB — 300MB) * (1.0–0.75) = 949MB

If you are not using this memory then you can change the *spark.memory.fraction* to reduce the user memory and give that space to spark memory. Normally I increase the *spark.memory.fraction to 85%, if I am not using user memory.*

Spark Memory: This is the memory pool which is used and managed by spark process. While memory optimization, we should optimize this part of the memory first.

Its size can be calculated as *("Java Heap" – "Reserved Memory") * spark.memory.fraction*, and with Spark 1.6.0+ defaults it gives us *("Java Heap" – 300MB) * 0.75*. For example, with 4GB heap this pool would be 2847MB in size. This whole pool is split into 2 regions – Storage Memory and Execution Memory, and the boundary between them is set by *spark.memory.storageFraction* parameter, which defaults to 0.5.

Storage Memory: This pool is used for both storing Apache Spark cached data, for temporary space serialized data "unroll", and "broadcast" variables. In case there is not enough memory to fit the whole unrolled partition it would

directly put it to the drive if desired persistence level allows this. As of "broadcast", all the broadcast variables are stored in cache with MEMORY_AND_DISK persistence level.

Execution Memory: This pool is used for storing the objects required during the execution of Spark tasks. For example, it is used to store shuffle intermediate buffer on the Map side in memory, also it is used to store hash table for hash aggregation step. This pool also supports spilling on disk if not enough memory is available, but the blocks from this pool cannot be forcefully evicted by other threads (tasks). This means -

- There is free space available in Storage Memory pool, i.e. cached blocks don't use all the memory available there. Then if Execution Memory required more space, it will just borrow the space from Storage Memory pool. It can also evict the less used object (cached) from execution memory block to HDD and make space for execution memory.
- Storage Memory pool size exceeds the initial Storage Memory region size and it has all this space utilized, then also it can not borrow space from execution memory, unless execution memory is free. If execution memory is full then this situation causes forceful eviction of the blocks from Storage Memory pool, unless it reaches its initial size.

Example 1: