databricks

PySpark

75. DATABRICKS & PYSPARK

SPARK SQL

Follow us on :

YouTube

LinkedIn

CDU

Cloud And Data Universe

SPARK SQL

- SPARK SQL is a Spark module for structured data processing.

- It provides a programming abstraction called as DataFrames and can also act as a distributed SQL query engine.

SPARK SQL

- It offers a SQL like query support, enabling a variety of users to easily query the data without using spark code.
- It can read data from variety of sources like csv, json, parquet, database(jdbc), etc.
- It is not a complete alternative to RDBMS, but it blurs the line between RDD and relational tables to great extent.

SPARK SQL

- Under the hood it uses catalyst optimizer to perform optimization in order to generate different spark plans.
- To use sql queries on Dataframes you create tempview on or managed/unmanaged tables, then simply fire sql queries on them as you query a table in any database.
- You are not limited to only SELECT queries, you can do a number of operations in DDL and DML.

CDU
Cloud And Data Universe

SPARK SQL

- Eases development for ones who already know SQL.
- You can also save the output of a query in a DataFrame and use it later.

databricks

PySpark

76. DATABRICKS & PYSPARK

SPARK MEMORY MANAGEMENT

Follow us on :

YouTube

LinkedIn

CDU

Cloud And Data Universe

JVM

- Spark is developed in Scala language.
- Scala runs on JVM.
- JVM stands for Java Virtual Machine.
- JVM is a virtual machine/computer where the execution takes place, basically the execution environment.

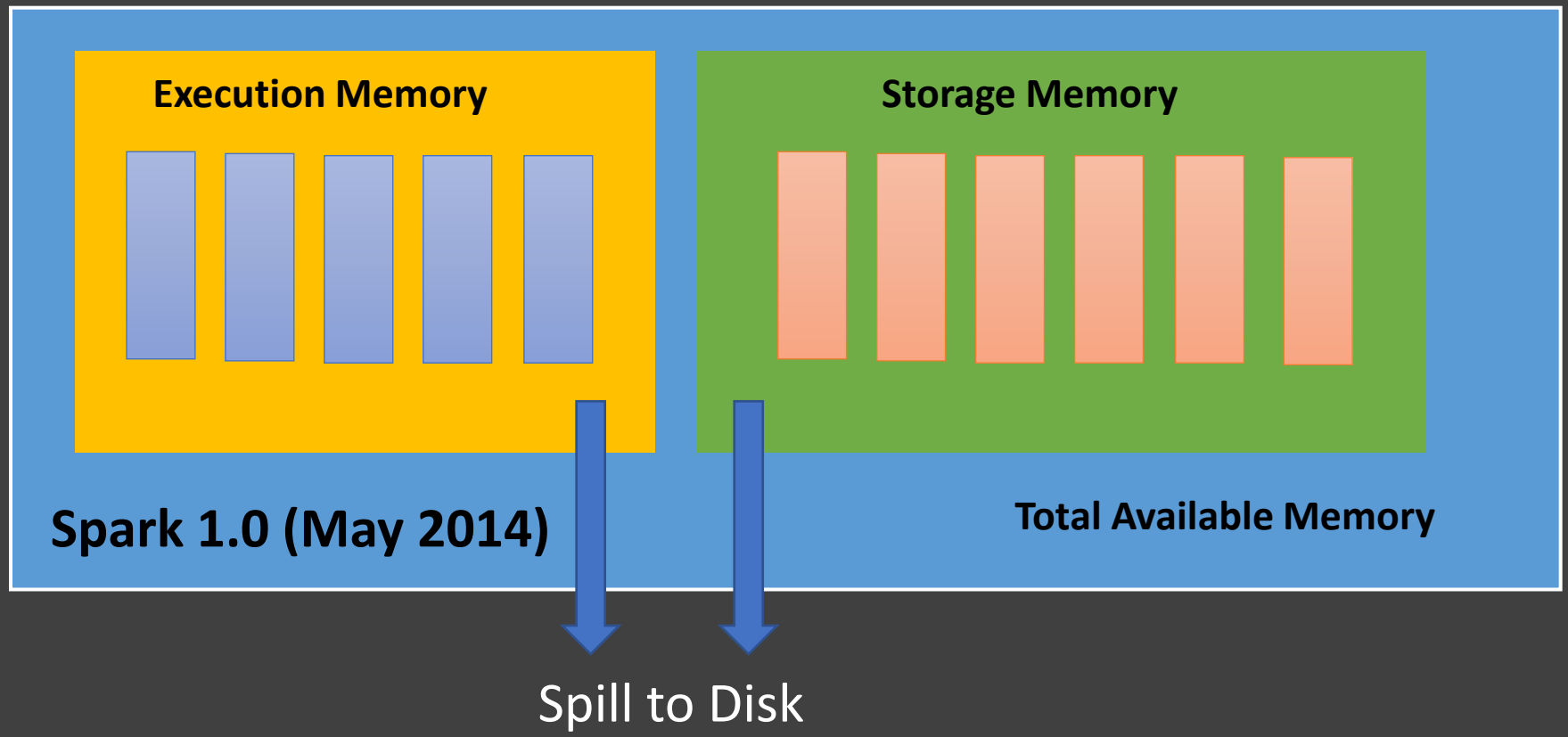
PYSPARK

- PySpark is a layer added on top of Scala, which enables to interact with spark in python language.

- The underlying execution will still be taking place on JVM.

CDU
Cloud And Data Universe

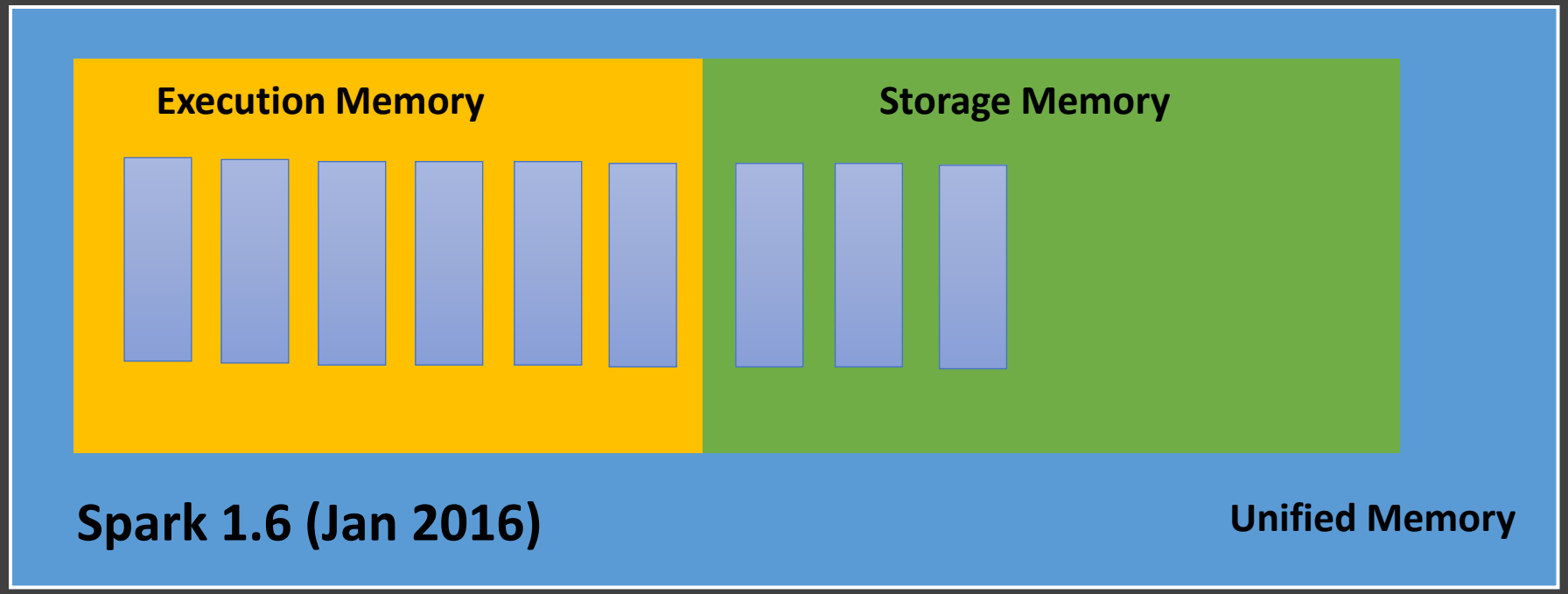
SPARK MEMORY



Cloud And Data Universe

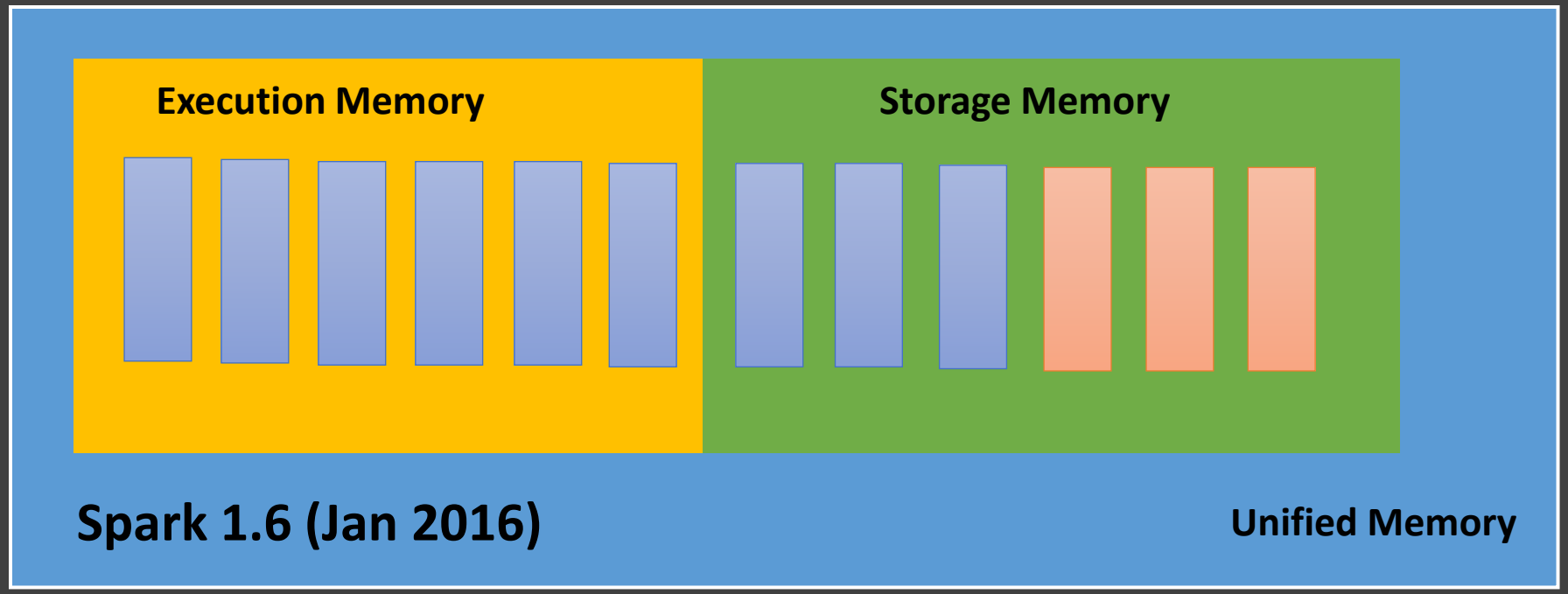
databricks PySpark

SPARK MEMORY



CDU
Cloud And Data Universe

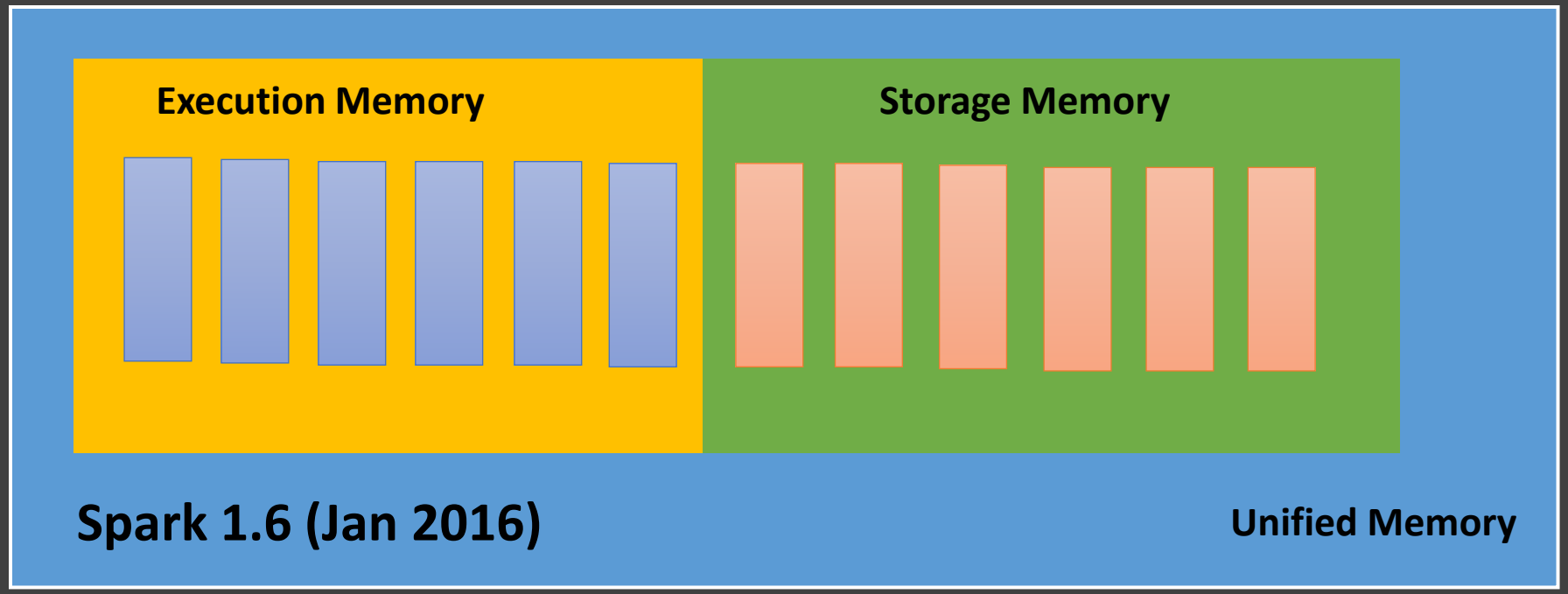
SPARK MEMORY



Evict using
LRU to disk

Cloud And Data Universe

SPARK MEMORY



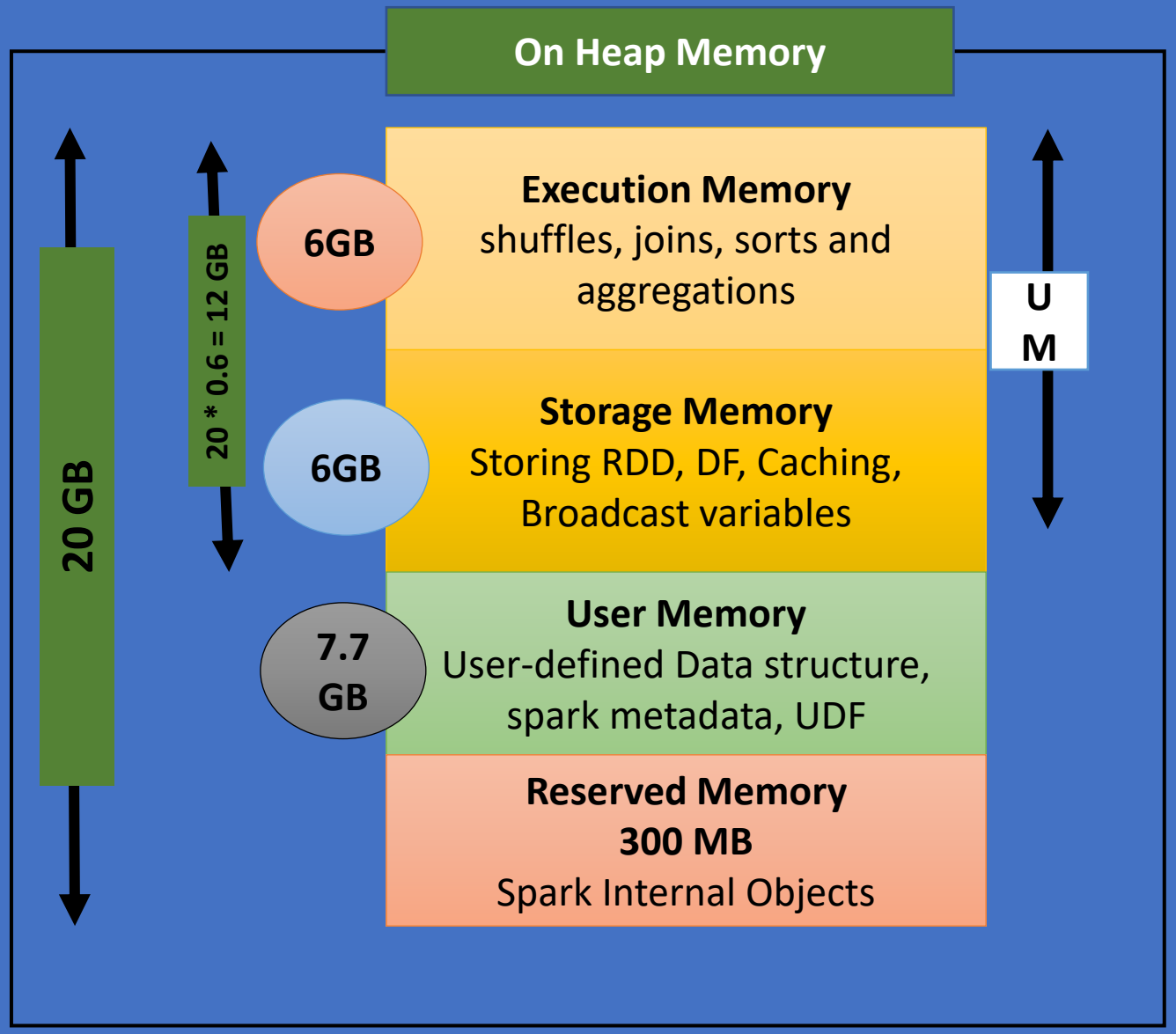
Evict using
LRU to disk

Cloud And Data Universe

Spark Memory Management

- Memory usage in spark broadly falls under 2 categories: execution and storage.
- Both shared a unified region.
- When spark doesn't use execution memory , storage can acquire all the available memory and vice versa.
- When execution memory is using storage memory, it can evict storage memory.
- When storage memory needs more memory, it cannot forcefully evict execution memory, it must wait for spark to release memory and then acquire it.

Executor



Off – Heap Memory
Caching

Overhead
(384 MB)

SPARK MEMORY MANAGEMENT

- Broadly memory is divided into on-heap memory and off-memory.
- On-heap memory is managed by JVM.
- Off-heap memory is managed by OS.

Properties and memory calculations

- spark.executor.memory (on-heap memory) Eg. 20 GB
- spark.memory.fraction (0.6) = 20 GB * 0.6 = 12 GB
- spark.memory.storage.fraction (0.5) = 12 GB * 0.5 = 6 GB.
- Hence, storage memory is 6GB and execution memory is 6GB.
- Reserved memory is 300 MB
- User memory = (20GB *0.4)-300mb = 7.7 GB

CDU
Cloud And Data Universe

Off-heap memory

- Off-heap memory is disabled by default.
- spark.memory.offheap.enabled =0
- Spark.memory.offheap.size = 1GB

Overhead memory

- Overhead memory by default is set 384 MB.
- However spark takes max(384 MB, 10% of spark.executor.memory).
- This memory is beyond the on-heap memory.
- Spark.executor.memoryoverhead

databricks PySpark

77. DATABRICKS & PYSPARK

GARBAGE COLLECTION

Follow us on :

YouTube LinkedIn

CDU
Cloud And Data Universe

Garbage Collection (GC)

- Garbage collection is one of the important concepts in any java application.
- As spark runs on JVM and is memory based it is must to know this GC process.
- What exactly is Garbage collection?

CDU
Cloud And Data Universe

Garbage Collection (GC)

- Memory is divided into two parts : On-heap and Off-heap.
- Spark executor uses on-heap memory for execution and storage.

Garbage Collection (GC)

- As spark is built to process large amount of data it will create large number of objects in memory , once the available memory is exhausted, it needs more memory, in such case when it runs out of memory the application will crash or become unresponsive leading to memory errors.
- Objects which are no longer used are termed as garbage and need to be cleaned. This process is termed as garbage collection!

CDU
Cloud And Data Universe

Garbage Collection (GC)

- This where garbage collection kicks in.
- Garbage collection is a process which automates freeing up/ releasing memory that is no longer in use by application.
- GC plays a critical role in ensuring spark has memory available to run the application efficiently.

Garbage Collection (GC)

- But, how does GC do it?
- As large number of objects are created in memory, these need to be removed timely so there is space for other objects.
- When JVM runs out of memory, it should pause the application and run a garbage collection cycle to free up memory.

Garbage Collection (GC)

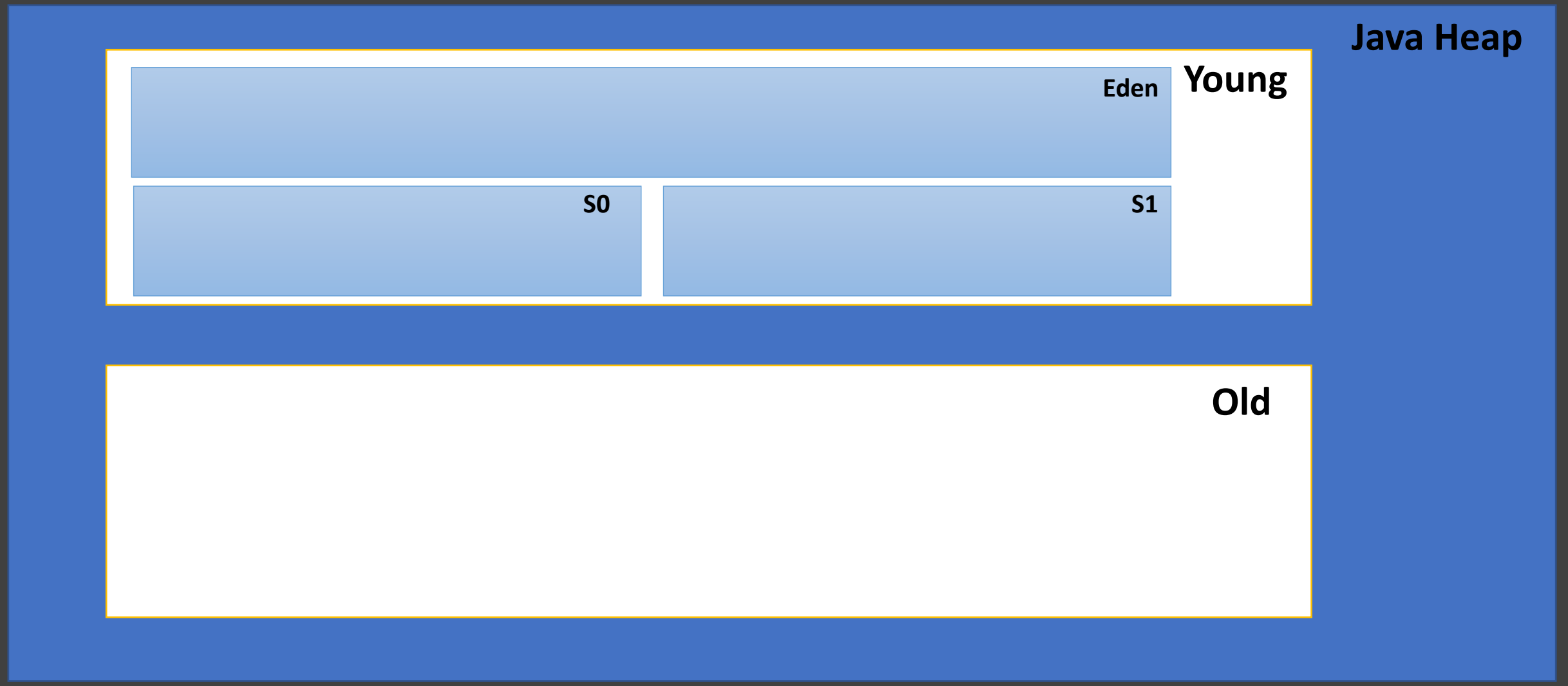
- Sounds simple, it isn't so!
- We need to tune GC at times when it isn't working as expected, but before we do it, we must understand further details about memory management in JVM.

databricks PySpark

Garbage Collection Process

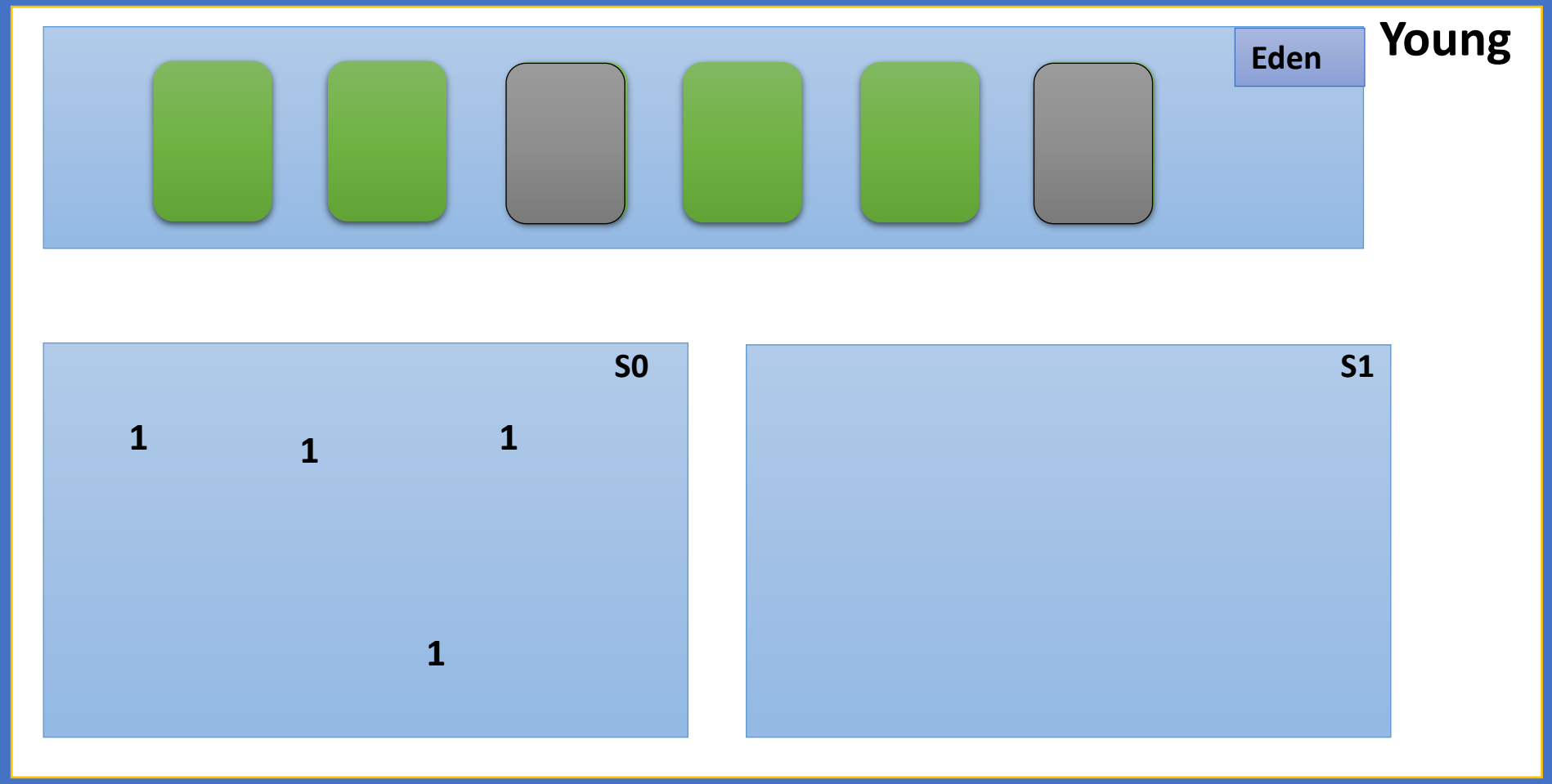
- Mark: GC goes through the object graph and marks the objects that are reachable as live, else as unreachable.
- Sweep: deletes unreachable objects.
- Compact: compacts the memory by moving around the objects.

CDU
Cloud And Data Universe



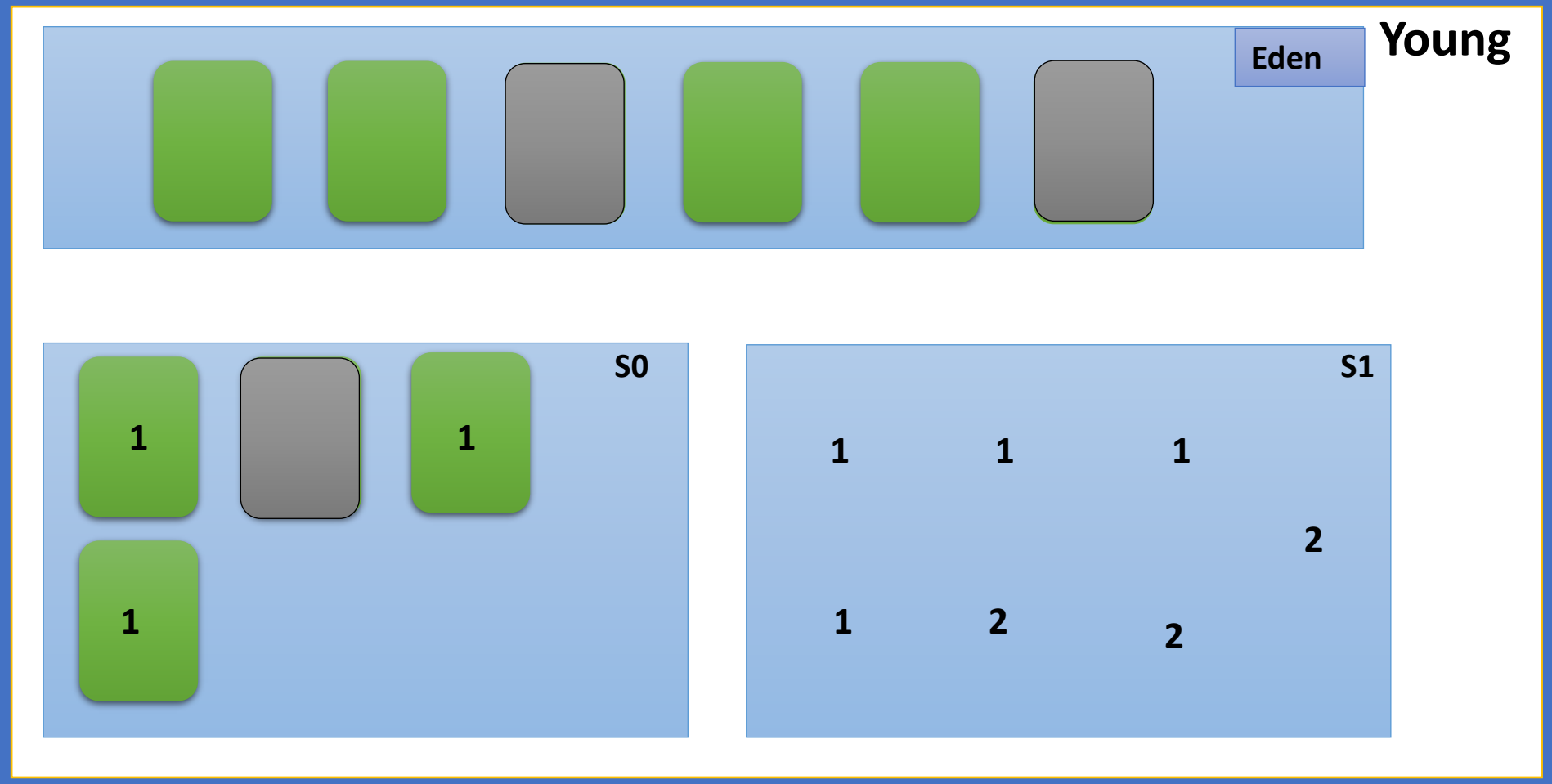
Cloud And Data Universe

Java Heap



Cloud And Data Universe

Java Heap



CDU
Cloud And Data Universe

Java Heap

- Java heap space is divided into two regions : Young and old.

- Young region/generation is for holding short-lived objects, Old region/generation is for holding objects with longer life.

- Young generation is divided further into Eden, Survivor1 and Survivor2.

CDU
Cloud And Data Universe

Minor GC – 1st Cycle

- At the beginning of execution of an application, newly created objects are loaded in Eden space. Over the time the objects acquire Entire Eden space and it Eden becomes full where no more space is available for newer objects. The allocation of new object fails here which triggers minor GC!

- The objects which are live(survived) are marked as reachable and are moved to next stage in Survivor 0 and the ones which are not are marked as unreachable and are swept and cleaned, thus completing 1st cycle of GC which is a minor GC.

- This now makes space for new objects in Eden.

Minor GC – 1st Cycle

- The survived objects are marked with a counter to indicate how many times it has survived the garbage collection!
- So after first minor gc, every survived object will marked as 1 since it survived first minor gc cycle.

Minor GC – 2nd Cycle

- Now application, assigns more objects in Eden region and it might be possible that some objects in Survivor 0 space become unreachable and some of them in Eden as well could be unreachable.

- Once Eden is full now, it needs more space but it will be unable to acquire new space and the allocation fails which will trigger another minor gc!

- Live Objects from Survivor 0 are moved to Survivor 1, live objects from Eden are moved to Survivor 1, unreachable objects in both Eden and Survivor 0 are removed.

Minor GC – 2nd Cycle

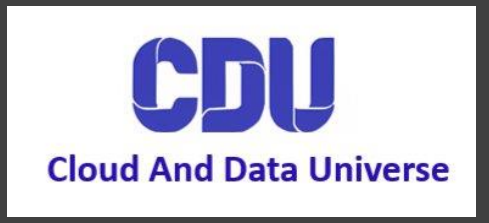
- Now, Survivor 1 holds objects which has survived 1 and 2 cycles of garbage collection and are marked as 1 and 2 respectively indicating the gc cycle count each one them survived.

Minor GC cycles

- This process of minor GC cycle continues until a threshold is hit.
- This threshold is defined by a parameter.
- Once the threshold is hit all the surviving objects are now moved to old generation where they have survived for longer time until now.

Old Generation and Major GC

- Over the period old generations reaches a state of becoming full and a thread(process) which keeps a watch on old generation alerts JVM.
- This is a significant step in GC as major GC is triggered!
- This where garbage collection is done across the heap on all the regions, by following mark, sweep , compact process!
- This can also cause the application to halt!



Garbage Collector

➢ **1. Serial Garbage collector:**

It is the simplest and oldest garbage collector in java.
It uses a single thread to perform GC.
Due to this approach, it is only suitable for single threaded or small-scale applications which limited memory.
It uses a “stop-the-world” (STW) approach , meaning the execution of application will be paused during GC process.
This can lead to pauses which are larger in duration for large applications.
Due to this application throughput decreases which is a drawback.

Garbage Collector

➢ **2. Parallel Garbage collector:**

Also known as throughput collector.

As the name suggests parallel garbage collector uses multiple threads for GC.

It divides the heap into smaller regions and uses multiple threads to perform GC at the same time(concurrently).

This reduces the STW pauses.

It is best suited for multi-core systems and applications that demand throughput.

CDU
Cloud And Data Universe

Garbage Collector

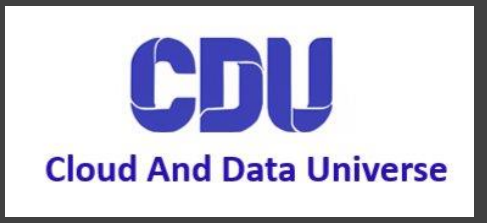
➢ **3. Concurrent Mark-Sweep (CMS) collector:**

CMS collector is designed to reduce pause times.

As the name suggests it performs most of work concurrently with the application threads.

It divides the GC into stages of concurrent marking and sweeping.

It significantly reduces pause times, however may not perform well on large heaps and can sometimes lead to fragmentation issues.



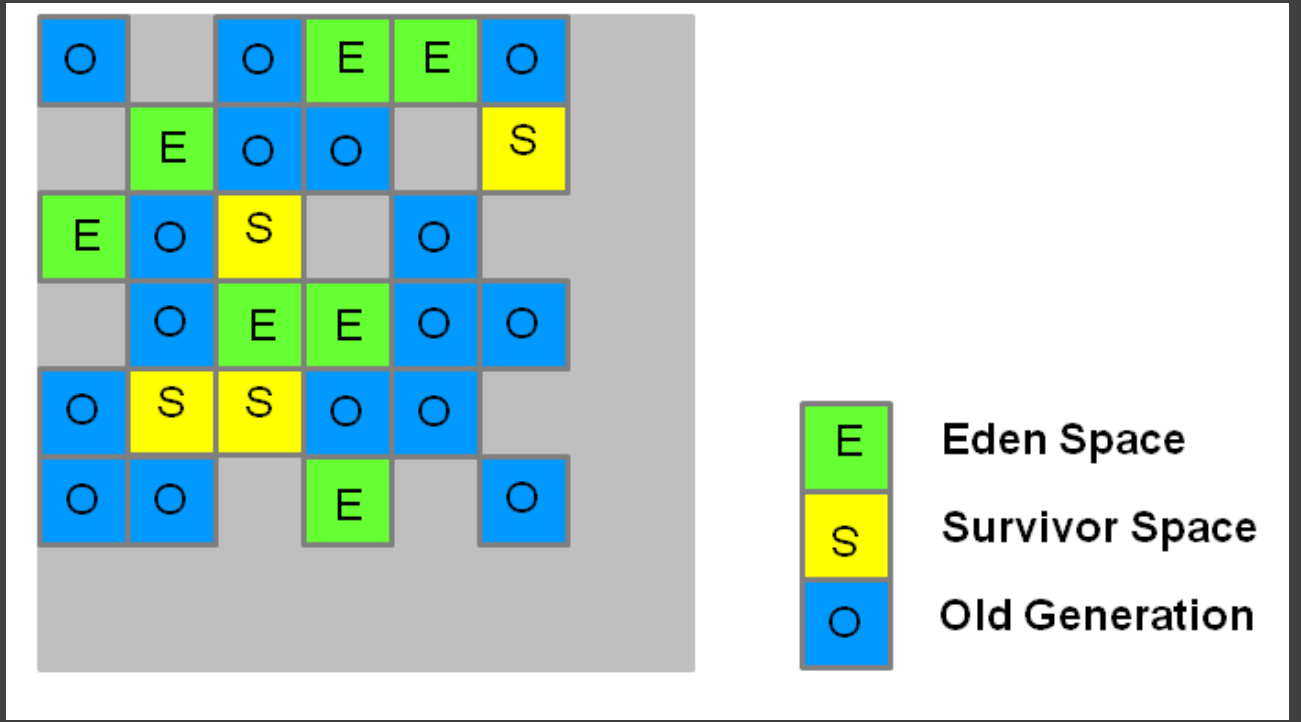
Garbage Collector

➢ **4. G1 (Garbage collector) collector:**

G1GC is a low-pause, server style garbage collector.
It uses concurrent and parallel phases to achieve its target pause time and to main high throughput.
When G1GC determines that garbage collection is necessary, it collects the regions with the least live data first(garbage first).
The G1GC divides the heap into equally sized regions, here the JVM sets the size. The region size varies from 1mb to 32mb depending on heap size. The region count is max 2048 . In this the eden, survivor and old generations are logical sets of these region and not contiguous.

databricks PySpark

G1GC heap allocation



CDU
Cloud And Data Universe

Garbage Collector

➢ **5. ZGC collector:**

It is designed to provide consistent and low latency performance.
It focuses on keeping pause time predictable even for very large heaps.
It uses a combination of techniques, including a compacting collector and read-barrier approach, to minimize pause times.
It is suitable for applications where low latency responsiveness is critical.

Garbage Collector

➢ **6. Shenandoah garbage collector:**

It is designed to provide low latency performance.

It focuses to minimize pause times.

It uses a barrier-based approach and uses multiple phases to perform concurrent marking, relocation and compaction.

It is designed for applications where ultra-low pause times are essential, even for very large heaps.

CDU
Cloud And Data Universe

databricks

PySpark

78. DATABRICKS & PYSPARK

CPU TERMINOLOGIES

Follow us on :

YouTube

LinkedIn

CDU

Cloud And Data Universe

CPU

- CPU is central processing unit.
- Also known as processor or micro processor.
- It is the brain of computer.
- CPU is responsible for processing all the work/tasks.

CPU Core

- CPU processes or executes a tasks.
- To perform this it needs a physical core!
- One core can handle only task at a time, hence a single core CPU can handle only one task at a time.
- As the evolution took place in chip industry, CPU manufacturers developed CPU chips with multiple cores providing more computations power and execution of tasks in parallel !

databricks PySpark

CPU Core

CPU Cores	Parallel Execution of tasks
1	1 task
2	2 tasks
4	4 tasks
6	6 tasks
8	8 tasks
16	16 tasks

CDU
Cloud And Data Universe

CPU Manufacturers

- Qualcomm
- Intel
- AMD
- NVIDIA

Multi-threading & Hyper-threading

- Multi-threading is a software process that allows a single process to run multiple threads at same time.
- Hyper-threading is a hardware process that allows a single physical core to run multiple threads at same time.
- Multi-threading can be used on any CPU, but Hyper-threading can be used on only CPU's which support it.

CDU
Cloud And Data Universe

Hyper-threading

- Hyper-threading makes a single physical core as two logical CPU's.
- This is a trick done to make operating system recognize every single CPU core has 2 CPU's!
- The benefit it provides is parallel execution!
- Due to this if one core is halted , other can execute the task.

databricks PySpark

Hyper-threading Cores

CPU Cores	Logical Processors
1	2
2	4
4	8
6	12
8	16

CDU
Cloud And Data Universe

databricks

PySpark

79. DATABRICKS & PYSPARK

DATABRICKS COMPUTE & CLUSTERS

Follow us on :

YouTube

LinkedIn

CDU

Cloud And Data Universe

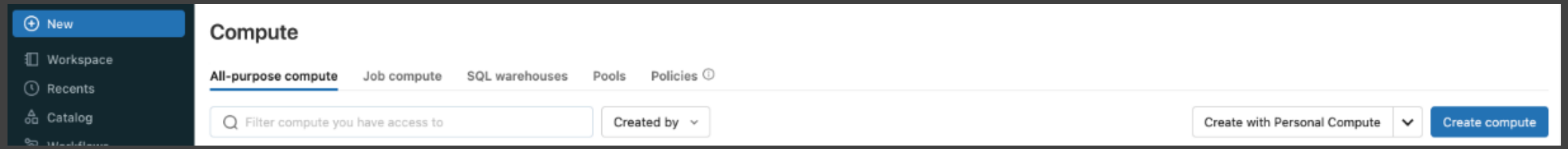
COMPUTE

- Compute is group of computing resources required to execute activities.
- In Databricks users need compute to run variety of tasks in data engineering, data science, data analytics, ETL , streaming and machine learning.

CDU
Cloud And Data Universe

databricks PySpark

COMPUTE TYPES



CDU
Cloud And Data Universe

COMPUTE TYPES

- **All Purpose Compute** : Used to run notebooks interactively. You can create cluster with single or multiple nodes, terminate and restart.

- **Job Compute** : Used to run automated jobs. The databricks job scheduler automatically creates a job compute whenever a job is configured to run on new compute. The compute terminates automatically when job is completed. You cannot restart the job compute.

CDU
Cloud And Data Universe

COMPUTE TYPES

- **Serverless Compute for notebooks:** On-demand, scalable compute used to execute SQL and Python code in notebooks.
- **Serverless Compute for Jobs:** On-demand, scalable compute used to run databricks jobs without configuring and deploying infrastructure.
- **SQL Warehouses:** Used to run SQL queries on data objects.

CDU
Cloud And Data Universe

CLUSTER TYPES

- **Single Node Cluster:**
- These consist of single machine which has one driver and it responsible for entire execution.

- These are suitable for learning and testing in databricks.

- These are cost friendly and good for beginners.

- As there is only one machine they have limited processing power and suitable for only small scale data processing needs.

CDU
Cloud And Data Universe

CLUSTER TYPES

- **Multi Node Cluster:**
- These contain multiple machines enabling parallel processing.
- As a result, these clusters can process huge amount of data.
- They offer great performance and scalability.
- Auto scaling feature enables to handle dynamic workloads.

CDU
Cloud And Data Universe

CLUSTER TYPES

- **High Concurrency Cluster:**
- These are designed to support simultaneous execution of tasks for multiple users.
- Best suitable for scenarios where multiple users need to collaborate and use resources at same time enabling efficient resource utilization.

CDU
Cloud And Data Universe

CLUSTER TYPES

- **GPU Enabled Cluster:**

- These are designed for machine learning and deep learning algorithms as they require large amount of processing power.

databricks

PySpark

80. DATABRICKS & PYSPARK

CLUSTER MANAGERS

Follow us on :

YouTube

LinkedIn

CDU

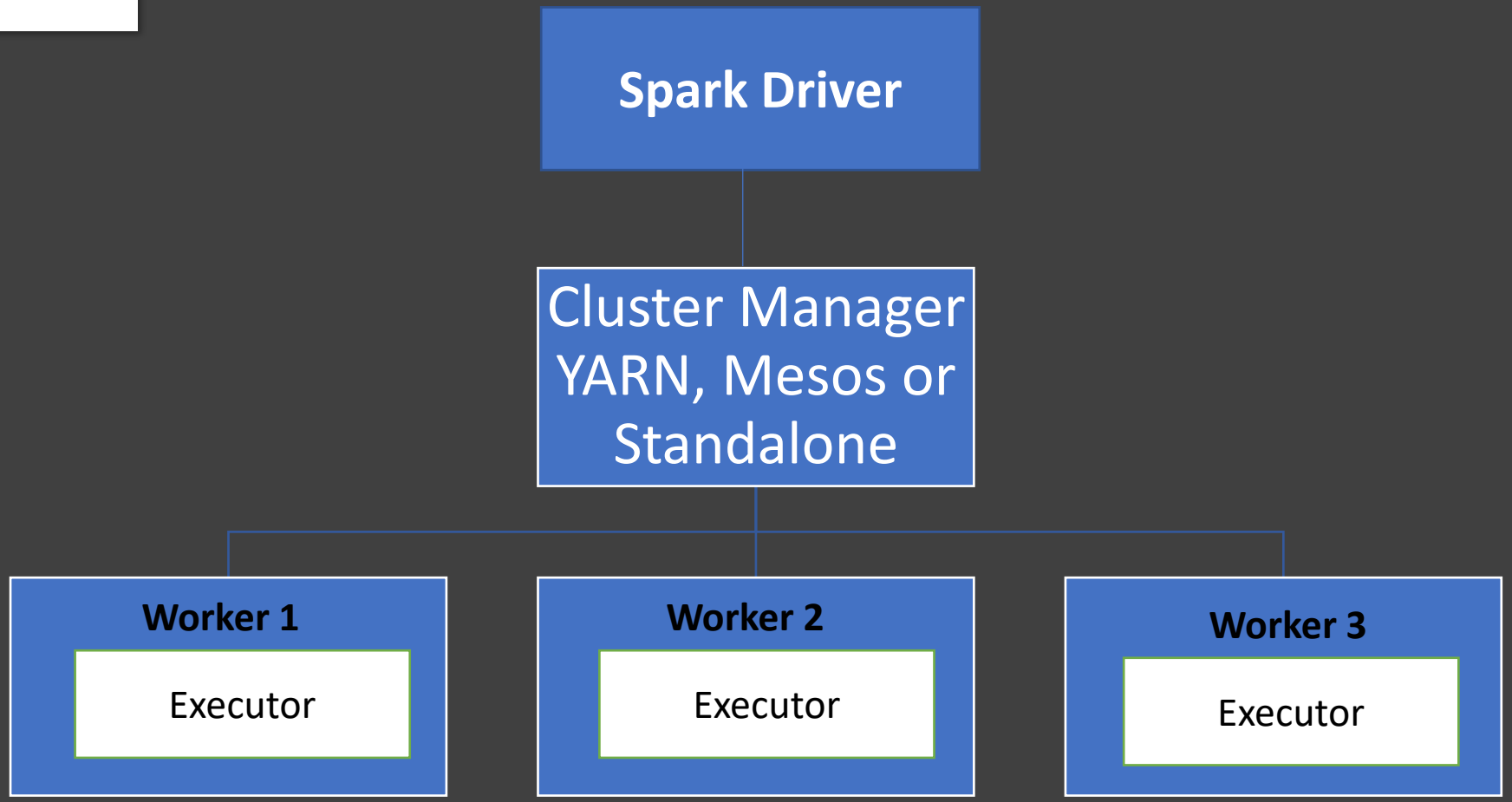
Cloud And Data Universe

CLUSTER MANAGER

- Spark is designed to efficiently scale up from one to hundreds to thousands of compute nodes.
- But, how does spark achieve this ?
- This is where cluster manager comes into picture.
- To achieve this flexibility spark uses either one of cluster managers : Hadoop YARN, Apache Mesos, Spark Standalone cluster.

databricks PySpark

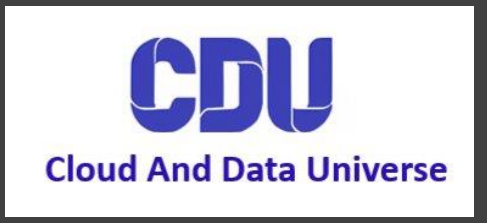
CLUSTER MANAGER



CDU
Cloud And Data Universe

SPARK ARCHITECTURE

- Spark uses master-slave architecture.
- Driver is the co-ordinator.
- Driver communicates with many number of worker nodes which have executors in them.
- Both driver and executors each have a separate java process.
- A driver and executors together are termed as spark application.



SPARK ARCHITECTURE

- Spark application is launched on a group of machines using cluster manager which is an independent service.
- Driver is the process where main() method of the program runs.
- It is the process running the user code that creates sparkcontext, creating RDD's and performing transformations and actions.

Cluster Manager

- Spark depends on cluster manager to launch executors.
- The cluster manager is a pluggable service/component in spark.
- This allows spark to run on top of different external managers like YARN and Mesos as well as built in Standalone cluster manager.

Spark-submit command

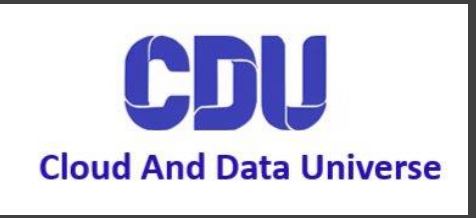
- Spark-submit command is a utility for executing or submitting spark jobs.
- These jobs can be submitted locally or to a cluster.
- Spark comes with a spark-submit.sh script for Linux, Mac and spark-submit.cmd command file for windows. These files are available in $SPARK_HOME/bin directory.

Spark-submit options

- Spark-submit options are parameters which can be specified when submitted a spark application.
- These options are important for configuring spark application as per the requirements including deploy mode, driver memory, executor memory, executor cores, etc.
- These options provide control on how the application is run and how the resources are used in spark cluster.

Spark-submit options

```
./bin/spark-submit \
  --master <master-url> \
  --deploy-mode <deploy-mode> \
  --conf <key<=<value> \
  --driver-memory <value>g \
  --executor-memory <value>g \
  --executor-cores <number of cores>  \
  --jars  <comma separated dependencies>
  --class <main-class> \
  <application-jar> \
  [application-arguments]
```



Deploy modes

- **Cluster mode:** The driver program runs independently of the client that submitted the job. Once the driver program is launched on a cluster node, the client disconnects and communicates with the cluster manager to request the resources and coordinate the execution of tasks.

- **Client mode:** Client machine maintains an active connection to cluster manager throughout the execution of spark application. This allows client to track the progress of application. In this mode the driver runs locally and all other executors run on the cluster.

Spark-submit options

- --master: The master URL for the cluster (e.g. spark://23.195.26.187:7077)
- --deploy-mode: cluster or client
- --conf: key-value pair (e.g. --conf <key>=<value> --conf <key2>=<value2>)

Driver and Executor resources

- **--driver-memory :** specify the driver memory. This can be increased when collecting large data.
- **--driver-cores** : number of cores to be used by spark driver.
- **--executor-memory** : specify the amount of memory to be used by executor
- **--executor-cores** : specify the number of cores to be used by executor.
- --total-executor-cores : total number of executor cores to use.

databricks PySpark

Different ways to submit spark jobs

```
#Run application locally on 8 cores
./bin/spark-submit \
  --class org.apache.spark.examples.SparkPi \
  --master local[8] \
  /path/to/examples.jar \
  100
```

```
# Run on a YARN cluster in cluster deploy mode
./bin/spark-submit \
    --deploy-mode cluster \
    --master yarn \
    --class org.apache.spark.examples.SparkPi \
    --executor-memory 20G \
    --num-executors 50 \
    /spark-home/examples/jars/spark-examples_versionxx.jar 1000
```

```
# Run on a Mesos cluster in cluster deploy mode
./bin/spark-submit \
  --class org.apache.spark.examples.SparkPi \
  --master mesos://207.184.161.138:7077 \
  --deploy-mode cluster \
  --executor-memory 20G \
  --total-executor-cores 100 \
  http://path/to/examples.jar \
  1000
```

```
# Running Spark application on standalone cluster
./bin/spark-submit \
  --class org.apache.spark.examples.SparkPi \
  --master spark://207.184.161.138:7077 \
  --deploy-mode cluster \
  --executor-memory 5G \
  --executor-cores 8 \
  /spark-home/examples/jars/spark-examples_versionxx.jar 1000
```

CDU
Cloud And Data Universe

databricks

PySpark

81. DATABRICKS & PYSPARK

RESOURCE ALLOCATION

Follow us on :

YouTube

LinkedIn

CDU

Cloud And Data Universe

Steps for deriving cores and memory

1. Find the total number of partitions.
2. Get the total number of cores needed to process above partitions.
3. Decide on maximum number of cores per executor.
4. Get total number of executors needed.
5. Calculate memory for each core.
6. Calculate memory for each executor.

CDU
Cloud And Data Universe

Calculating cores and memory

- There is a 20gb file which needs to be processed and we need to determine the total number of cores and memory required to process it.

1. Total no. of partitions: 20gb = 20480mb -> 20480/128 = 160 partitions
2. Total cores required for max parallelism = 160 cores
3. Max cores per executor (recommended 5) = 5 cores
4. Total executors needed = 160/5 = 32 executors

CDU
Cloud And Data Universe

Calculating cores and memory

- 5. Memory for each core (1.5 times reserved memory of 300mb) = 450mb ~ 512mb

- 6. Memory for each executor = 512mb * 5 = 2560mb

Time estimation

- 1 partition / task / core / slot

- One task takes 4 mins to run, how long will it take for entire 20gb file to be processed?

- -> 4 mins, considering max parallelism.

Driver Memory

- Usually can assign or choose 4gb.
- In case of collect need to be careful, if the data returned to driver is more than memory size, it will lead to out of memory exception.

Resource management

- 10 nodes and each node has 16 cores and 64gb memory(RAM).

1. On each node 1 core and 1gb memory is required for operating system on internal processes. So now we have 15 cores and 63gb memory.

Resource management

2. Choose number of cores: Considering 5 as an optimum no. of tasks that run concurrently in an executor we perform below calculations:

- 5 cores per executor.
- Total no. of cores = 10 * 15 = 150 cores
- Total no. of executors = 150/5 = 30 executors (less 1 executor for AM).
- Executors per node = 29/10 ~ 3 executors

Resource management

- Memory per executor -> 63gb/3 = 21gb
- Overhead memory -> max(384mb, 21 * 0.1) = max(384mb, 2.1gb) = 2.1gb
- Final memory allocated to each executor -> 21gb - 2.1gb = 18.9gb

databricks

PySpark

82. DATABRICKS & PYSPARK

DYNAMIC ALLOCATION

Follow us on :

YouTube

LinkedIn

CDU

Cloud And Data Universe

Static Allocation

- In Spark, Static allocation pre-allocates resources to an application before it starts running.
- The amount of resources is fixed during the runtime of application and can't be changed.
- In cases where application needs more resources it could take longer to run or could even fail due to shortage of resources.

CDU
Cloud And Data Universe

Dynamic Allocation

- Dynamic allocation is a feature in spark which enables spark to increase or decrease the number of executors dynamically depending on load.
- As a result, this provides optimization during the execution to scale up and down the executor count.
- As needed spark dynamically adds executors during the execution and removes them when not needed.

CDU
Cloud And Data Universe

Dynamic Allocation

- When is this suitable?

1. The resource required for processing tasks is unknown or keeps changing.
2. The load is changing during the execution of job like the size or volume.

Dynamic Allocation

- When is this suitable?

1. The resource required for processing tasks is unknown or keeps changing.
2. The load is changing during the execution of job like the size or volume.

Dynamic Allocation

- How does it work?
- The application initially starts with certain number of executors which are pre-defined.
- During the execution if there are pending tasks spark adds more executors subject to being available. (Scale-Up)
- If executors are idle for a significant amount of time, Spark will remove them. (Scale-down)

CDU
Cloud And Data Universe

Dynamic Allocation Configs

- **spark.dynamicAllocation.enabled:** Set to true to enable dynamic allocation and set to false to disable it.

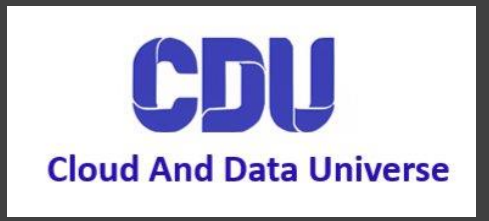
- **spark.shuffle.service.enabled:** Needs to be set to true to enable it. Shuffle operations like joins, groupby need to save the data. This service preserves the shuffle files written by executors e.g. so that executors can be safely removed.

Dynamic Allocation Configs

- **spark.dynamicAllocation.initialExecutors:** The initial number of executors when spark application starts execution.

- **spark.dynamicAllocation.minExecutors: Minimum number of executors to keep running the spark application. When workload decreases, less executors are required, this will ensure only minimum no. of executors are running and will also help in reducing costs.**

- **spark.dynamicAllocation.maxExecutors: Maximum n. of executors which spark application can acquire.**



Dynamic Allocation Configs

- spark.dynamicAllocation.executorIdleTimeout: As the executors which as idle need to be removed. This config to be set in seconds decides it. If the executor is idle for more than this duration it is removed. This also helps to reduce costs.

databricks

PySpark

83. DATABRICKS & PYSPARK

SERIALIZATION & DESERIALIZATION

Follow us on :

YouTube

LinkedIn

CDU

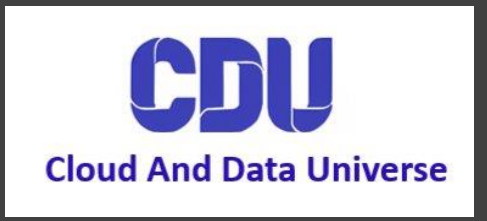
Cloud And Data Universe

Serialization

- Data is stored in form of objects like rdd, dataframe and data structures like list, set, dictionary.
- These objects are JVM objects.
- These objects need to be transferred from one node to other in scenarios where shuffle is needed.
- This transfer takes place through network.

Serialization

- However, the JVM objects cannot be transferred through the network and must be converted into a supported format to be transferred through the network.
- The process of converting these JVM objects into byte stream is called as Serialization !
- Once the objects are serialized they can be transferred over the network.



Serialization

- Another need for serialization is when data needs to be spilled to disk.

Deserialization

- When the transfer takes place, the byte stream is again converted into JVM objects which is called as deserialization !

Serialization Implementation

- There is difference when it comes to implementing serialization in Scala-spark and PySpark.
- Scala-spark uses java serialization by default.
- There is also an option of kryo serialization which is faster compared to java serialization.

Serialization Implementation

- In Pyspark, the default serialization library is Pyrolite.
- PySpark also supports custom serializers for transferring data which can provide significant performance improvement.
- By default PySpark uses the PickleSerializer, which leverages Python's cPickle serializer to serialize almost any Python object.
- PySpark also provides an option to use other serializers like MarshalSerializer which can be faster, but supports fewer data types.