* Spark performance optimization/tuning is performed with multiple aspects/perspective such as code level optimization, spark configuration, select right file format and its compression, choosing right spark default optimization features and hardware configuration.
* ---------------------------------------CODE----------------------------------------------------------------------------
* Shuffle operations:

1. Repartition vs coalesce

\*\*recommended to use coalesce

1. groupByKey, reduce byKey

\*\* recommended to use tree reduce by key and give no of hops instead of directly merging all outputs of executor to driver

1. co group and join (join operations)
2. shuffle can be controlled by: config properties

* spark.shuffle.compress
* spark.shuffle.file.buffer
* spark.shuffle.service.enabled

1. transformation
2. Narrow
3. Wide – needs shuffling

\*\* recommended to use narrow transformation

* ----------------------------------------------Hardware------------------------------------------------------------
* Spark is a in memory processing system but there are disk writes too. Executors have disks and disk writes can occur during

1. Shuffle
2. Spills to disks while shuffling/redistributing data to executors
3. Caching/persistence to disk
4. Determining factors:

* Disk
* Cores
* Memory
* Network speed
* Code optimization to get best out of the hardware config

1. Check spark UI for the jobs, stage, executor, SQL, storage, DAG, task – quartile, data skew
2. Minimize data scans: Reduce amount of data while making any transformation

* By Partitioning
* Bucketing – not much recommended. Should be used only as per use case
* Partition Pruning
* Filtering
* Avoid Data Skewness – salting
* Join optimization as it needs shuffling
* Set threshold while performing joins
* Avoid using unnecessary counts/repartition, distinct count (instead we can use approx. count), as these operations needs to scan whole data
* We can use persisting or caching – reusing data – faster subsequent actions -spark cache is fault tolerant (too many caches create GC overhead, possible slowdown and disk spills)
* Use of broadcast variable for read only variable for caching
* Accumulators
* Use seq.par.foreach instead to achieve parallelism – disadvantage can have race conditions and non-deterministic results, use accumulators and synchronization methods
* To avoid UDF

1. Since its custom code – spark default optimizations are not applicable
2. UDF operates at each row of a dataset makes its very costly operation
3. Each row deserialization and apply custom logic to each row makes it very costly and non-optimized

* --------------------------------------------------FILE FORMAT-----------------------------------
* File Format + file Compression
* Spark.rdd.compress
* -------------------------------------------------PARALLELISM-------------------------------------
* Increase in parallelism can considerably increase performance
* Config property

1. Spark.default.parallelism
2. Spark.sql.files.min.PartitionNum

* --------------------------------------------------GARBAGE COLLECTION(GC)-----------------------------
* GC details are available in SPARK UI/logs
* Worker logs
* -verbose:gc -XX:+PrintGCDetails -XX:PrintGCTimeStamps to the java options
* -------------------------------------------Serialization-----------------------------------------------------------
* Recommended Kryo serializer – default is java serializer
* Kyro is much more efficient than default java serialization
* Config property:
* Conf.set(“spark.serializer”,”org.apache.spark.serializer.KryoSerializer”)
* -----------------DYNAMIC ALLOCATION for Multitenant Environment------------------------
* Dynamic allocation is good for Multitenant environment
* Config properties:

1. Spark.dynamicAllocation.enabled
2. Spark.dynamicAllocation.minExecutors
3. Spark.dynamicAllocation.maxExecutors
4. Max Executors
5. Spark.dynamicAllocation.executoridleTimeout

* \_------------------------------------------Executors and off heap settings ---------------------------
* Num executors
* --num-executors (spark submit equivalent)
* (Spark.executors.instances, “16”)
* Executor cores
* --executors.cores (spark submit equivalent)
* (Spark.executors.cores, “5”)
* Executor memory
* --executors-memory (spark submit equivalent)
* (Spark.executor.memory, “2g”)
* Off heap memory : slower than onheap memory but gives extra space beyond onheap memory and free from garbage collect so we avoid garbage collection overhead
* (Spark.memory.offHeap.enabled, true)
* (Spark.memory.offHeap.size, “5g”)
* Driver memory
* Spark.driver.memory

\*\* ./bin/spark-submit –help

* ---------------------------------Disk Spills------------------------------------------
* For shuffle – disk spill occurs
* Config property to control disk spills:
* Spark.shuffle.file.buffer
* Spark.shuffle.compress
* Spark.shuffle.spill.compress
* Spark.io.compression.snappy.blocksize