SPARK OPTIMIZATION CHEATSHEET, V1

Prepared by: Subhasish Ghosh, Aug2021

	Issue(s)	Solution(s)	Code Sample
1	High processing cycles.	Principle: "Filter-Early, Filter-Often". Use filter() & map() effectively *before* shuffle ops (e.g., reduceByKey() & groupByKey()). Us them *before* & *after* join() ops.	Use filter() transformations to remove unneeded records. Use map() transformations to project only required fields in RDD.
2	Poor performance in associate operations (e.g., sum() & count())	If you group by a key on a partitioned OR distributed dataset exclusively with the aim of aggregating values for each key, then reduceByKey() is *always* better than using groupByKey().	rdd.map(lambda x: (x[0],1)) \ .groupByKey() \ .mapValues(lambda x: sum(x)) \ .collect() → Poor code rdd.map(lambda x: (x[0],1)) \ .reduceByKey(lambda x,y:x+y) \ .collect() → Good code
3	Poor performance in associate operations (e.g., sum() & count())	If you group by a key on a partitioned OR distributed dataset exclusively with the aim of aggregating values for each key but inputs and outputs to your reduce function are different, then combineByKey() is *always* better than using groupByKey(). Internally, combineByKey combines values of a PairRDD partition by applying an aggregate function.	comb_rdd = student_rdd.map(lambda t: (t[0], (t[1], t[2]))) \ .combineByKey(createCombiner, mergeValue, mergeCombiner) \ .map(lambda t: (t[0], t[1][0]/t[1][1]))
4	Poor performance in associate operations (e.g., sum() & count())	Using foldByKey() is *always* better than using groupByKey() when you wish to aggregate values based on keys and wish to perform an associative operation providing a zero value.	val maxByDept = employeeRDD.foldByKey(("dummy",0.0)) ((acc,element)=> if(acc._2 > element._2) acc else element)

5	Memory issues on Spark Workers caused mostly owing to excess Network I/O because of poor design choice; i.e., passing too much data to a function.	Use broadcast method to create a broadcast variable(s) which are shared across Workers using an efficient peer-to-peer sharing protocol, based on BitTorrent. 5 benefits: 1) no shuffle, 2) highly scalable peer-to-peer distribution, 3) replicate data once/Worker instead of once/Task, 4) tasks reuse, 5) provides serialized objects, so they are efficiently read always.	massive_list = [......] \ def big_func(x): # function with massive list \ rdd.map(lambda x: big_func(x)).saveAsTextFile → Poor code
			stations = sc.broadcast(sdata) \ status = sc.textFile('file:///opt/spark/data/mydata/status') \ → Good code
			.map(lambda x: x.split(',')) \
			.keyBy(lambda x: x[0])

6	Memory exceptions during collecting data owing to poor design choice.	Best practice is: 1) use take(n) & takeSample() *always* if you need to just inspect instead of collect() & take(). 2) if doing ETL, then persist to a filesystem OR a database instead of doing collect() & take().	
7	Lack of parallelism.	At app level OR using spark-defaults.conf is spark.default.parallelism setting. Best practice here is to have this value equal to OR 2x number of cores on each Spark Worker. You will need to tweak for your own environment, to see what works the best.	Formula to use: spark.default.parallelism = spark.executor.instances * spark.executors.cores * 2

8	Spark Executors idle for long periods of time.	Use dynamic allocation which releases back Executors back to the cluster pool if they are idle for a specified period. Should be implemented typically as a system setting to help maximize system resources.	# enable DynamicAllocation disabled by default
			spark.dynamicAllocation.enabled = True \ spark.dynamicAllocation.minExecutors=n \ spark.dynamicAllocation.maxExecutors=n
			# upperbound is by default set to 60 secs

9	Poor partitioning design choice.	Best practice #1: Avoid small files resulting in too many small partitions. This results in massive overhead of spawning these tasks which is greater than processing required to execute the small tasks.	E.g., filter() ops on partitioned RDD may result in some partitions smaller than others. Solution is to filter() ops with a repartition() OR coalesce() and specify a number less than input RDD. NOTE: repartition() can increase OR reduce number of partitions; coalesce() can only reduce the number of partitions.
10	Poor partitioning design choice.	Best practice #2: Avoid exceptionally large partitions. E.g., loading an RDD from a large compressed file using an unsplittable compression format such as Gzip.	4 solutions: 1) Avoid using unsplittable compression, if possible. 2) Uncompress file locally (e.g., /tmp) before loading into RDD. 3) Repartition *before* a large shuffle operation. 4) Repartition immediately *after* the first transformation against the RDD.
11	Poor partitioning design choice.	Best practice #3: Avoid having fewer partitions than Executors.	Recommended practice is to make this an input parameter to your application and then test out with different input values and optimize for performance. No silver bullet; needs to be found with trial & error.
12	Wrong result in Accumulators.	Accumulators typically used for ops purposes. If used in add-in-place ops to calculate results inside a map() ops, then results can be wrong. Best practice is to always use accumulators only within actions computed by Spark driver, e.g., foreach() action.	Using *custom* accumulators for accumulating vectors as either lists OR dictionaries. from pyspark import AccumulatorParam class VectorAccumulatorParam(AccumulatorParam): ... vector_acc = sc.accumulator({..........................}, VectorAccumulatorParam())
13	Actions result in frequent re-evaluations.	Caching an RDD persists data in memory; same routine can then reuse it multiple times when subsequent actions are called without requiring any explicit re-evaluation. Cached partitions are stored in Mem of Executor JVMs on Spark Worker nodes.	words = doc.flatMap(lambda x: x.split()) \ .map(lambda x: (x,1)) \ .reduceByKey(lambda x,y:x+y) words.cache() \ words.count() # triggers computation \ words.take(2) # no computation \ words.count() # no computation
14	Actions result in frequent re-evaluations.	persist() more optimized than cache(). Offers additional storage options including MEMORY_AND_DISK, DISK_ONLY, MEMORY_ONLY_SER, MEMORY_AND_DISK_SER, and MEMORY_ONLY (same as cache() method). Additionally, persist can use replication to persist the same partition on more than one node.	words = doc.flatMap(lambda x: x.split()) \ .map(lambda x: (x,1)) \ .reduceByKey(lambda x,y:x+y) words.persist()
15	Stack overflow issues and long lineage leading to long recovery times.	Using checkpointing eliminates the need for Spark to maintain RDD lineage, especially applicable for streaming and/or iterative processing applications. Checkpointing is expensive so happens only after an action is called.	words = doc.flatMap(lambda x: x.split()) \ .map(lambda x: (x,1)) \ .reduceByKey(lambda x,y:x+y) words.checkpoint() \ words.count() \ words.isCheckpointed() \ words.getCheckpoinFile()