Contents

- Understanding Spark UI
- Troubleshooting spark application

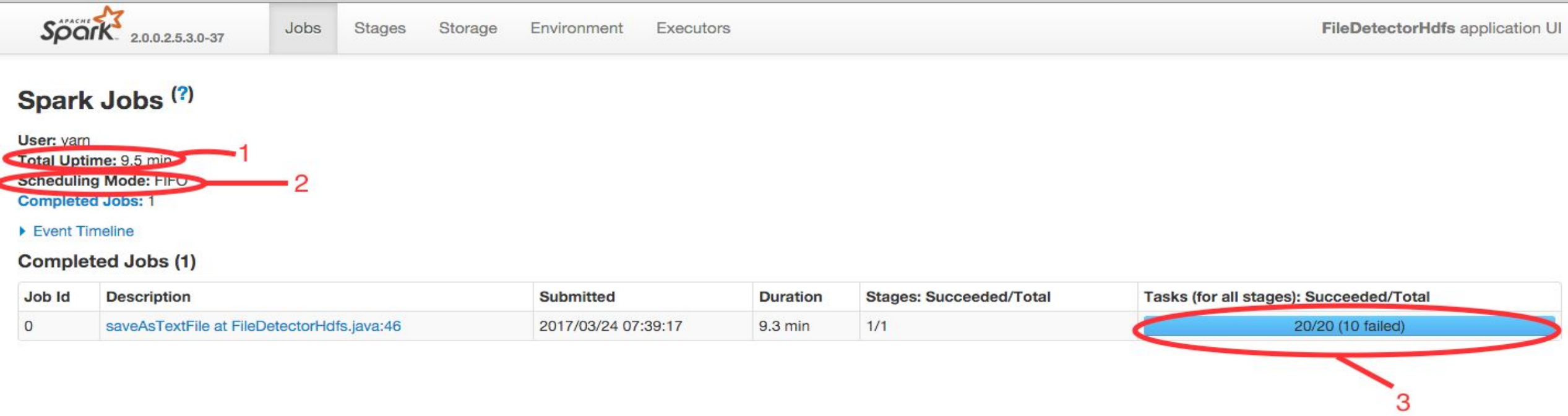
Understanding Spark UI

Relationship between jobs, stages, tasks, partitions, actions and transformations

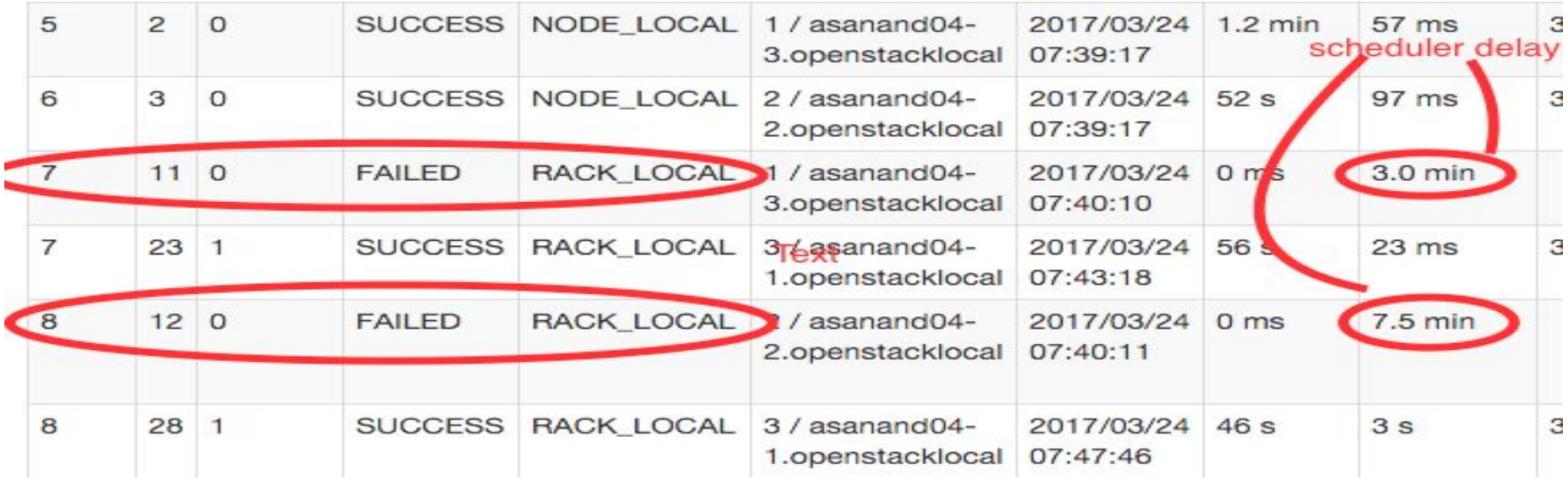
- Number of *jobs* = Number of *actions* in the application
  - What are actions ? http://spark.apache.org/docs/latest/programming-guide.html#actions
- A job consists of a set of *stages*. Lets say we have a lineage graph of RDD a -> b -> c . Here RDD b is parent of RDD c and a is parent of b. In simplest case , the scheduler outputs a computation stage for each RDD in this graph. (In more complex case a set of RDD may be grouped to form a stage.)
- A stage consists of a number of *tasks*. A task is created for each *partition* in the RDD that belongs to the stage.
- More about the relationships between these in a separate write up.

Screenshots in this write up is taken from application id 1490270133337_0297. This can be found by logging to http://172.22.119.200:8080/ . User id and password is admin

Spark UI : Jobs



- (1) Total uptime: Total uptime is for application (not for jobs). Total uptime is sum of duration of all jobs + other things (may be AM container launch time)
- (2) Scheduling mode: The scheduling mode between JOBS submitted to the same SparkContext. Options are FIFO and Fair (round robin). Here we have only one job so FIFO seems to be better option.
- (3) Tasks (for all stages): Total number of tasks. 20/20 (10 failed) means 20 tasks were present and all 20 completed successfully but 10 attempts failed. So total there were 30 attempts for 20 tasks. See below picture (taken from stages section of UI)



5	2	0	SUCCESS	NODE_LOCAL	1 / asanand04-3.openstacklocal	2017/03/24 07:39:17	1.2 min	57 ms	3
6	3	0	SUCCESS	NODE_LOCAL	2 / asanand04-2.openstacklocal	2017/03/24 07:39:17	52 s	97 ms	3
7	11	0	FAILED	RACK_LOCAL	1 / asanand04-3.openstacklocal	2017/03/24 07:40:10	0 ms	3.0 min	
7	23	1	SUCCESS	RACK_LOCAL	3 / asanand04-1.openstacklocal	2017/03/24 07:43:18	56 s	23 ms	3
8	12	0	FAILED	RACK_LOCAL	2 / asanand04-2.openstacklocal	2017/03/24 07:40:11	0 ms	7.5 min	
8	28	1	SUCCESS	RACK_LOCAL	3 / asanand04-1.openstacklocal	2017/03/24 07:47:46	46 s	3 s	3

In above picture 5 and 6 succeeded in a single attempt while 7 and 8 took 2 attempts to succeed. Notice that locality level of failed tasks are RACK_LOCAL. If data and the code that operates on it are together then computation tends to be fast. Spark tries to execute tasks as close to the data as possible to minimize data transfer (over the wire). There are several levels of locality based on the data's current location. In order from closest to farthest:

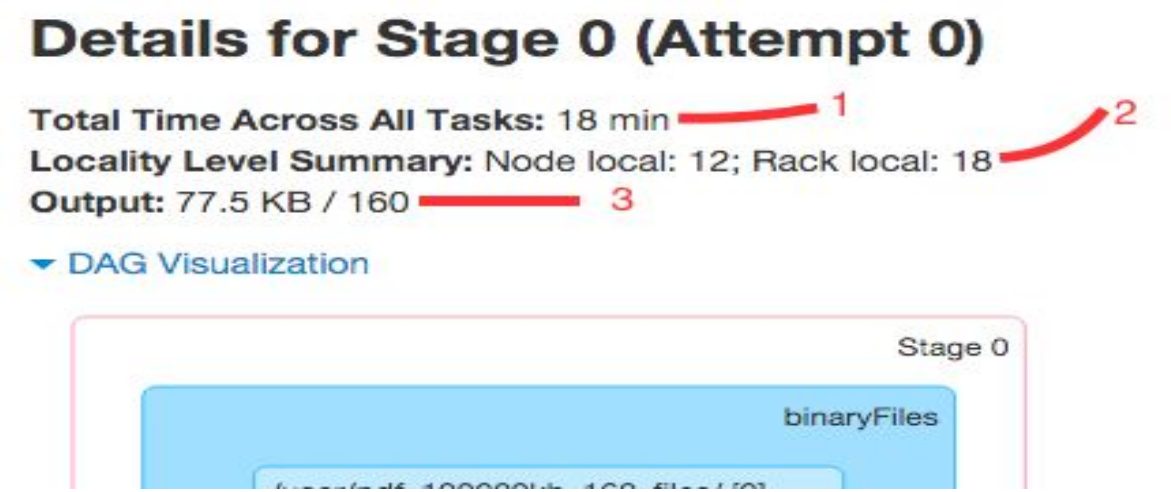
- PROCESS_LOCAL: data is in the same JVM as running code.
- NODE_LOCAL: data is on the same node. E.g. in HDFS on the same node. Slower than PROCESS_LOCAL.
- NO_PREF: no locality preference
- RACK_LOCAL: data is on a different server on the same rack so needs to be sent over network.
- ANY: data is elsewhere and not in same rack.

The wait timeout for fallback between each level is configured using spark.locality.wait (default value 3s). RACK_LOCAL performs bad , as data has to be transferred over network , specially when the size of partition is quite large. In fact all the 10 attempts which failed has the locality level of RACK_LOCAL. Many times this is

one of the main reason for scheduler delay (grey color line in event timeline). ***This can be solved by having small size partitioned RDD and more number of cores. The same application (new id 1490796291720_0008) when ran with 40 partitions, 10 cores per executor and 24g executor memory the time taken is 2.5 min (originally it was 9.5 min). Max scheduler delay is 0.1 s (vs 3 s originally).***

Note: The scheduler delay column shows 7.5 min and 3.0 min. But these include the running time of failed executor also (not just scheduler delay). In reality the max scheduler delay is just 3 s. Main culprit of such an exaggerated scheduler delay is OOMError caused by GC overhead limit exceeded or java heap space. Most of the time giving enough executor memory makes everything work fine.

Spark UI: Stages (Part 1)



- (1) Total time across all tasks: If we sum the duration of all tasks it will be around total time across all tasks. However many of the tasks run in parallel so the overall time is significantly low (its 9.5 min).
- (2) Locality Level summary : locality level is already discussed. Out of 30 attempts 12 were NODE_LOCAL and 18 were RACK_LOCAL.
- (3) output: 77.5 kb of data was produced while processing 160 records (each pdf file is one record).

Spark UI : Stages (part 2)

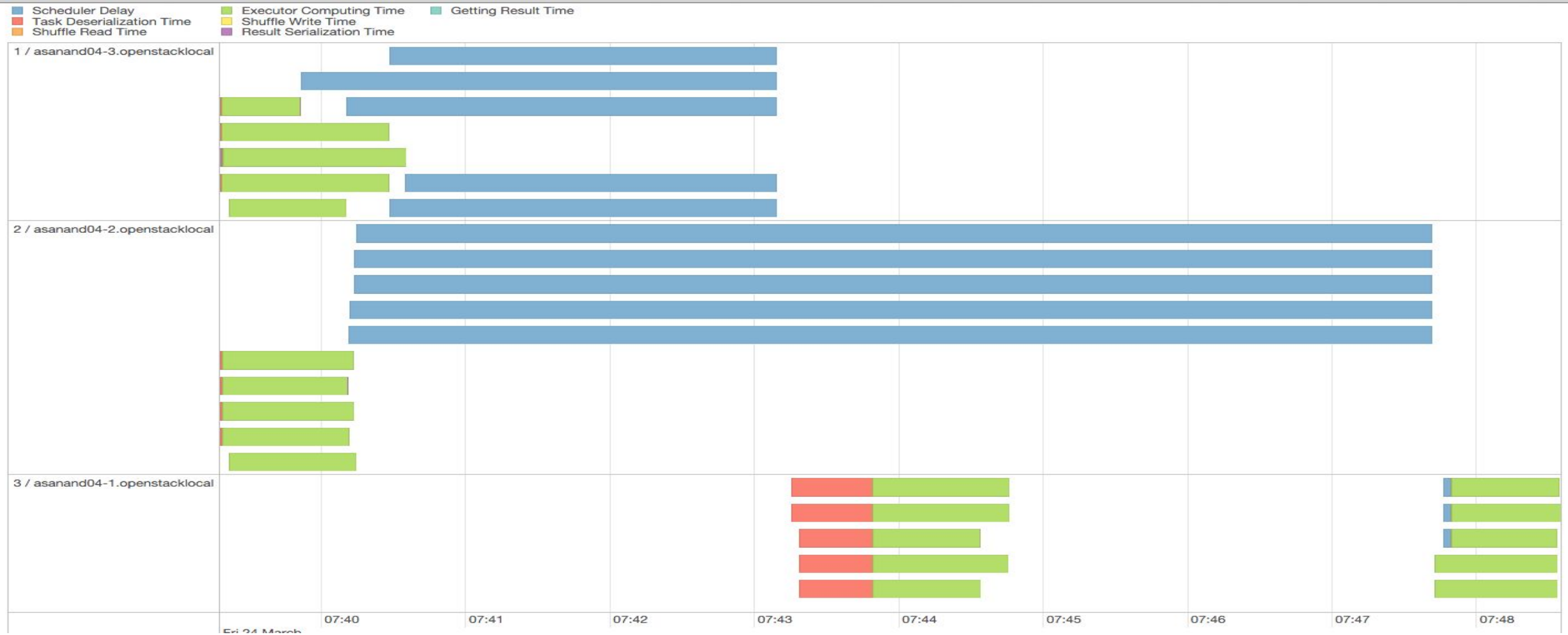
Tasks (30)

Index	ID	Attempt	Status	Locality Level	Executor ID / Host	Launch Time	Duration	Scheduler Delay	Task Deserialization Time	GC Time	Result Serialization Time	Output Size / Records	Errors
0	0	0	SUCCESS	NODE_LOCAL	1 / asanand04-3.openstacklocal	2017/03/24 07:39:17	33 s	0.1 s	0.6 s	11 s	1 ms	0.0 B / 8	
1	1	0	SUCCESS	NODE_LOCAL	2 / asanand04-2.openstacklocal	2017/03/24 07:39:17	55 s	65 ms	0.9 s	32 s	1 ms	0.0 B / 8	
2	8	0	SUCCESS	RACK_LOCAL	1 / asanand04-3.openstacklocal	2017/03/24 07:39:21	49 s	26 ms	17 ms	15 s	0 ms	11.1 KB / 8	
3	9	0	SUCCESS	RACK_LOCAL	2 / asanand04-2.openstacklocal	2017/03/24 07:39:21	53 s	24 ms	30 ms	32 s	0 ms	11.1 KB / 8	
4	10	0	FAILED	RACK_LOCAL	1 / asanand04-3.openstacklocal	2017/03/24 07:39:51	0 ms	3.3 min	0 ms		0 ms	0.0 B / 0	ExecutorLostFailure (executor 1 exited caused by one of the running tasks) Reason: Executor heartbeat timed out after 125952 ms
4	21	1	SUCCESS	NODE_LOCAL	3 / asanand04-1.openstacklocal	2017/03/24 07:43:15	57 s	0.1 s	34 s	39 s	0 ms	0.0 B / 8	

- (1) Index , Id and attempt: Partition 0 was processed by Task id 0 and was successful in attempt 0. However, partition 4 was processed by task id 10 which was not successful in attempt 0. It was attempted again with task id 21 and succeeded this time.
- (2) Locality level is already discussed.
- (3) Launch time: Using launch time we can easily identify a task in event timeline and examine the parameters of a task pictorially in event timeline. (event timeline is discussed next).

- (4) Scheduler delay: “Scheduler delay includes time to ship the task from the scheduler to the executor, and time to send the task result from the executor to the scheduler. If scheduler delay is large, consider decreasing the size of tasks or decreasing the size of task results.“ Some of the factors which affects Scheduler delay are : Locality level , waiting time for communications between the driver and the executor (particularly when driver and executor are on different machines), all resources are occupied etc.
- (5) Task deserialization time: The time taken to ship the jars from the driver to the executor. The very first set of tasks (on a executor) takes the maximum amount of task deserialization time (on that executor).
- (6) GC time: Not sure but seems its included in “duration” of a given task. Comparison between two applications with almost similar delays other than GC (e.g. scheduler, deserialization etc.) is covered later in this write up. Application which took longer to run had longer GC. Allocating more executor memory seems to lower this delay.
- (7) output size/ records: Each pdf file is treated as a record. Each task processes 8 records.

Spark UI : Stages (Part 3)



There are 3 executors

- Executor 1: There are 5 long scheduler delays. All the five tasks had locality level of RACK_LOCAL. By looking into logs , it was found that all the five tasks failed because of ExecutorLostFailure.

Executor heartbeat timed out after 125952 ms. Soon after that they were scheduled on executor 3. (can be solved by increasing spark.network.timeout ??)

- Executor 2: There are 5 long scheduler delays. All the five tasks had locality level of RACK_LOCAL. By looking into logs , it was found that all the five tasks failed because of OOMError which happened during container launch. Later these were scheduled on executor 3. Only way seems to solve it is by either increasing executor memory or optimizing application.

Note: Event timeline shows the whole failed task as scheduler delay but actually its not just scheduling delay. Scheduling had happened and executor was running and then after sometime some error occurred in executor. This is evident if we see Summary metrics table where max scheduler delay is only 3 s (as per event timeline it should be 7.5 min).

Impact of GC time: comparison

	Launch Time	Duration	Scheduler Delay	Task Deserialization Time	GC Time
ıl	2017/03/24 07:22:33	17 s	5 s	0.8 s	10 s
ıl	2017/03/24 07:22:33	20 s	0.1 s	0.8 s	9 s
ıl	2017/03/24 07:22:37	17 s	33 ms	29 ms	9 s
ıl	2017/03/24 07:22:37	17 s	2 s	0.8 s	10 s
ıl	2017/03/24 07:22:54	29 s	14 ms	19 ms	22 s
ıl	2017/03/24 07:22:33	17 s	5 s	0.8 s	10 s
ıl	2017/03/24 07:22:33	20 s	89 ms	0.8 s	9 s
ıl	2017/03/24 07:22:54	27 s	14 ms	19 ms	21 s
ıl	2017/03/24 07:22:54	29 s	14 ms	30 ms	22 s
ıl	2017/03/24 07:22:54	29 s	14 ms	22 ms	22 s
ıl	2017/03/24 07:22:33	17 s	5 s	0.8 s	10 s
ıl	2017/03/24 07:22:33	20 s	87 ms	0.8 s	9 s

Launch Time	Duration	Scheduler Delay	Task Deserialization Time	GC Time
2017/03/24 07:19:24	27 s	26 ms	26 ms	20 s
2017/03/24 07:19:24	48 s	18 ms	22 ms	10 s
2017/03/24 07:19:51	45 s	11 ms	21 ms	38 s
2017/03/24 07:19:20	45 s	52 ms	0.5 s	6 s
2017/03/24 07:19:20	30 s	92 ms	0.8 s	20 s
2017/03/24 07:19:51	32 s	27 ms	16 ms	26 s
2017/03/24 07:19:51	45 s	13 ms	13 ms	38 s
2017/03/24 07:19:51	45 s	13 ms	15 ms	38 s
2017/03/24 07:19:20	51 s	55 ms	0.5 s	10 s
2017/03/24 07:19:20	30 s	61 ms	0.8 s	20 s
2017/03/24 07:19:52	44 s	9 ms	9 ms	38 s
2017/03/24 07:20:05	23 s	12 ms	12 ms	16 s

The above given applications App1 (left) and App2 (right) are both processing 80 100000 kb files. App1 duration is 59 sec while App2 duration is 84 sec. Their scheduler delay and task deserialization time is almost comparable. Their duration seems to be mainly impacted by GC time. GC time for App2 is higher then App1.

Spark UI: Environment

Environment

Runtime Information

Name	Value
Java Home	/usr/jdk64/jdk1.8.0_77/jre
Java Version	1.8.0_77 (Oracle Corporation)
Scala Version	version 2.11.8

Spark Properties

Name	Value
spark.history.kerberos.keytab	none
spark.driver.host	172.22.119.204
spark.history.fs.logDirectory	hdfs:///spark2-history/
spark.eventLog.enabled	true
spark.ui.port	0
spark.driver.port	41062
spark.driver.extraLibraryPath	/usr/hdp/current/hadoop-client/lib/native:/usr/hdp/current/hadoop-client/lib/native/Linux-amd64-64
spark.yarn.queue	default
spark.yarn.historyServer.address	asanand04-1.openstacklocal:18081
spark.yarn.app.id	application_1490270133337_0297
spark.app.name	FileDetectorHdfs
spark.scheduler.mode	FIFO
spark.driver.memory	20g
spark.history.kerberos.principal	none
spark.executor.id	driver
spark.yarn.app.container.log.dir	/grid/0/hadoop/yarn/log/application_1490270133337_0297/container_e02_1490270133337_0297_01_000001
spark.submit.deployMode	cluster
spark.master	yarn
spark.ui.filters	org.apache.hadoop.yarn.server.webproxy.amfilter.AmIpFilter
spark.history.provider	org.apache.spark.deploy.history.FsHistoryProvider
spark.executor.extraLibraryPath	/usr/hdp/current/hadoop-client/lib/native:/usr/hdp/current/hadoop-client/lib/native/Linux-amd64-64
spark.executor.memory	20g

Environment tells us various configuration used with this spark application.

Spark UI: Executors

Spark 2.0.0.2.5.3.0-37 | Jobs | Stages | Storage | Environment | Executors

Executors

Summary

	RDD Blocks	Storage Memory	Disk Used	Cores	Active Tasks	Failed Tasks	Complete Tasks	Total Tasks	Task Time (GC Time)	Input	Shuffle Read	Shuffle Write
Active(3)	0	0.0 B / 31.5 GB	0.0 B	10	0	0	10	10	11.1 m (5.7 m)	0.0 B	0.0 B	0.0 B
Dead(2)	0	0.0 B / 21.0 GB	0.0 B	10	0	0	20	20	1.02 h (4.7 m)	0.0 B	0.0 B	0.0 B
Total(5)	0	0.0 B / 52.5 GB	0.0 B	20	0	0	30	30	1.20 h (10.3 m)	0.0 B	0.0 B	0.0 B

Executors

Executor ID	Address	Status	RDD Blocks	Storage Memory	Disk Used	Cores	Active Tasks	Failed Tasks	Complete Tasks	Total Tasks	Task Time (GC Time)	Input	Shuffle Read	Shuffle Write	Logs
4	asanand04-3.openstacklocal:41213	Active	0	0.0 B / 10.5 GB	0.0 B	5	0	0	0	0	0 ms (0 ms)	0.0 B	0.0 B	0.0 B	stdout stderr
3	asanand04-1.openstacklocal:49205	Active	0	0.0 B / 10.5 GB	0.0 B	5	0	0	10	10	11.1 m (5.7 m)	0.0 B	0.0 B	0.0 B	stdout stderr
2	asanand04-2.openstacklocal:57779	Dead	0	0.0 B / 10.5 GB	0.0 B	5	0	0	10	10	41.9 m (2.7 m)	0.0 B	0.0 B	0.0 B	stdout stderr
1	asanand04-3.openstacklocal:38893	Dead	0	0.0 B / 10.5 GB	0.0 B	5	0	0	10	10	19.2 m (2.0 m)	0.0 B	0.0 B	0.0 B	stdout stderr
driver	172.22.119.204:45564	Active	0	0.0 B / 10.5 GB	0.0 B	0	0	0	0	0	0 ms (0 ms)	0.0 B	0.0 B	0.0 B	stderr stdout

To get more info about the executor and tasks which ran on that executor we can follow the logs link provided in Executor view of Spark UI.

Learnings:

- Most of the failure occurred for locality level RACK_LOCAL. (Why ?)
- Its not the scheduler delay which contributes most to the total delay (its just a tiny fraction). Most of the delay is caused by failed executors (increase memory to solve it). Other factor which contributes most to delay is GC time (how to lower it ?).
- Parameter which affects execution significantly is executor memory. It can make execution fast (may be GC overhead decreases) and it also reduces number of failures.
- Sometimes the given application takes more time as there are some other unfinished application running on yarn. we can use “yarn application -list” to find the unnecessary jobs and kill them using “yarn application -kill application_id”. One of the application which took unexpectedly long time (42 mins) can be explained using this.

Troubleshooting spark application

- ExecutorLostFailure: Executor heartbeat timed out.

Solution: Try increasing value of spark.network.timeout . Default value is 120s. Keep increasing it until you do not see this exception. This should solve the problem but if application stops progressing after some tasks then look into logs if there is OOMError. If yes then high chances that you need to allocate more memory or change GC policy.

- All the failures occurring at RACK_LOCAL . is there a way to avoid RACK_LOCAL ?

Solution: There are two ways

(i) set spark.locality.wait.node to very high value. It won't go beyond locality level NODE_LOCAL. Default value is 3s. In our application it did not work. Application was hanged after processing some tasks.

(ii) set —num-executors equal to number of nodes in HDFS cluster. For spark-tika it reduced the RACK_LOCAL to 8 in worst case (and 0 in best case). Average duration of application on 16GB data was found to be 1 min with no failures.

Settings used were:

executor-memory 24g to 13g (won't work below 13g) ,

number of executors = 5 (= number of nodes in cluster),

number of partitions (= number of tasks) = 40.

Application was launched using:

time spark-submit --class FileDetectorHdfs --master yarn --deploy-mode cluster --driver-memory 6g --executor-memory 14g --num-executors 5 --executor-cores 10 --queue default jars/tmpvar/spark-tika.jar /user/pdf_100000kb_160_files /user/root/output 40

Other factor which may have helped to get such a low duration is less load on executor (in terms of memory and processing power) due to increased number of executors.

- Application is getting unexpectedly slow. If there are lots of OOMError or Full GC is called many times or OldGen is close to being full (chances are all 3 will exists together) in logs then use following ways to solve it.

Solution: There are two ways

(i) Lower the fraction of spark.memory.fraction and increase the number of partitions . Default value is 0.6 . The rest of the space (40%) is reserved for user data structures, internal metadata in Spark, and safeguarding against OOM errors in the case of sparse and unusually large records. Try to find out the right combination of spark.memory.fraction and number of partitions.

(ii) Try G1GC garbage collector. It can improve performance in some situations where garbage collection is a bottleneck. For our application on 16GB data it reduced the amount of executor memory requirement to 11g (below 11g it was not working). Result and other configs are same as point (2)-(ii) .

Application was launched using:

time spark-submit --class FileDetectorHdfs --master yarn --deploy-mode cluster --driver-memory 6g --executor-memory 24g --num-executors 4 --executor-cores 10 --conf "spark.executor.extraJavaOptions=-XX:+UseG1GC -XX:+PrintGCDetails -XX:+PrintGCTimeStamps" --queue default jars/tmpvar/spark-tika.jar /user/pdf_100000kb_160_files /user/root/output 40

- Application is not making any progress (look into incomplete application section in Spark UI history to confirm this).

Solution: It is highly likely that some other application is using all the resources. use “yarn application -list” to find such application. Use “yarn application -kill application_id” to kill that application.

Few Other things to keep in mind

- RAM size of nodes in my cluster is 32GB each and node memory for yarn is set to 28 GB. Sometimes (I saw it rarely) spark(or yarn ?) creates two executors on the same node. Now if my executor-memory is 24g then the next time if spark tries to create a new executor on the same node then the new executor will fail as OS won't be able to allocate enough memory to the new executor. This also causes delay in execution.
- I have decreased the driver memory to 6g as in our case the driver is not participating in computation and neither doing .collect or .take or .takeSample etc. . Our application terminates on distributed output action i.e. .saveAsTextFile. Hence driver does not need much memory (have not tested the lower limit yet).
- Use different settings for different amount of data ??
- The easiest way to increase performance I found was to increase the memory as much as we can.

Pendings

- I found in all cases number of tasks is equal to number of partitions. But how does spark create partition for binary files is still a puzzle to me.
- SPARK_WORKER_INSTANCES is set to 1 in spark2-env . Then also sometimes it allocates more than one exec on same node. Need to figure out why ?