COMP5349 – Cloud Computing

Week 7: Spark Distributed Execution

Dr. Ying Zhou
School of Computer Science

THE UNIVERSITY OF SYDNEY

Last Week

- Last week we covered distributed execution of MapReduce jobs
  - GFS and its open source counterpart HDFS
  - YARN resource management system
- All systems we used so far have master/slave architecture
  - One node(process) takes the coordinator role with global knowledge
  - This is different to the master/slave copy structure you might have encountered in database
- Potential issues with single master design
  - Single point of failure
  - Master as bottle neck
- Common ways to deal with such issues
  - Master maintains small amount of data and the important data maintained by master are replicated in other external nodes
  - Minimizes master’s involvement in a cluster’s usual business

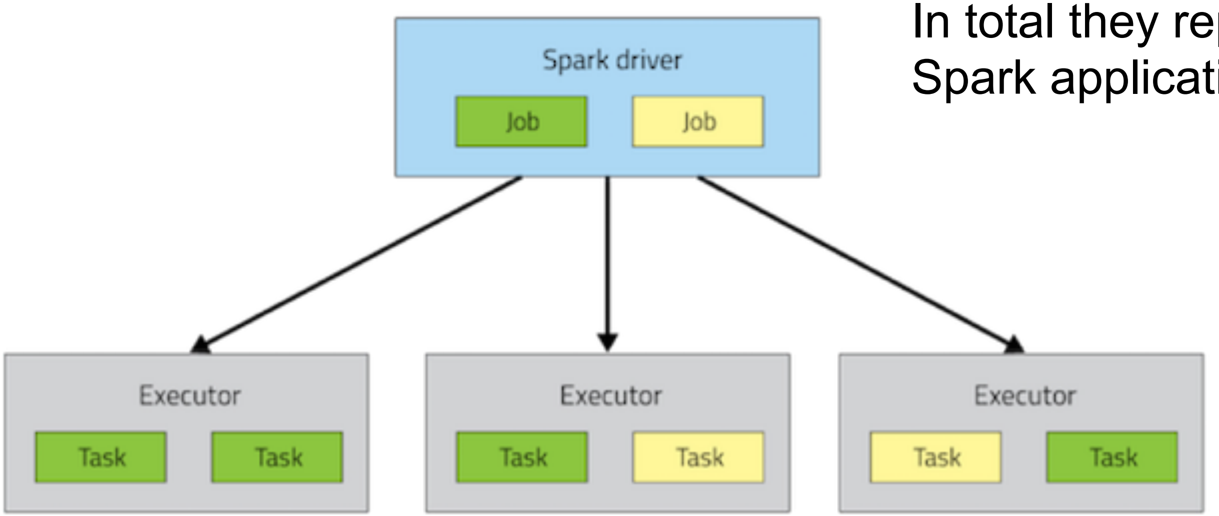
Last Week

- YARN basic components
  - Resource Manager, Node Manager, Application Master
- MapReduce Execution by YARN
  - Each Mapper/Reducer occupies a container with configurable resources allocated
  - The Application Master works out the number of containers required and request those through YARN Resource Manager
  - YARN RM would grant AM where (nodes) to use what amount of resources
  - AM would launch container through node manager and decide to run which mapper/reducer on which container
    - Each container runs a JVM executing mapper/reducer logic
    - Data locality preference would be applied at this level

Outline

- **Spark Execution View**
  - **Application**
  - **Job**
  - **Stage**
  - **Task**
- **Impact of Shuffling**
- **Cluster Resource Specification**

How Spark Execute Application

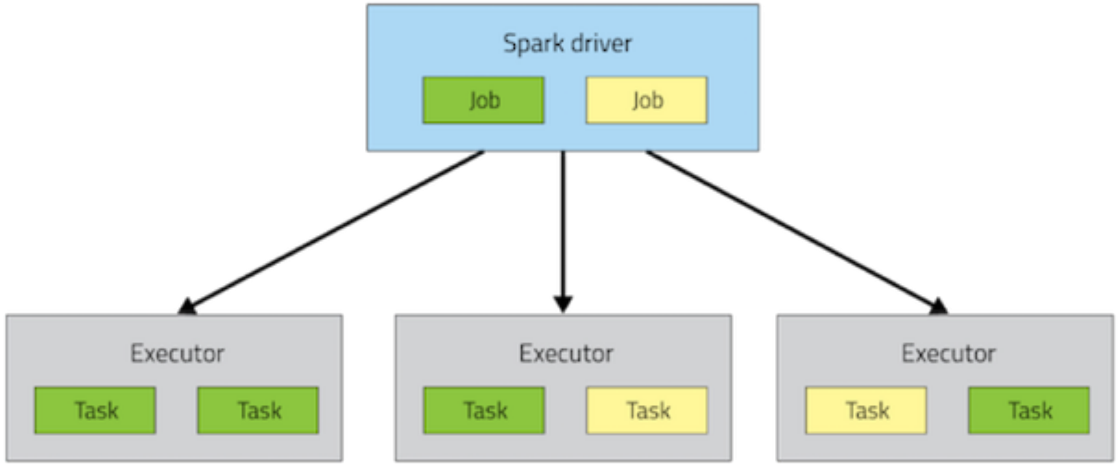


In total they represent a Spark application

"The **driver** is the process that is in charge of the high-level control flow of work that needs to be done. The **executor** processes are responsible for executing this work, in the form of ***tasks***, as well as for storing any data that the user chooses to cache. Both the **driver** and the **executor**s typically stick around for the entire time the application is running"

Based on http://blog.cloudera.com/blog/2015/03/how-to-tune-your-apache-spark-jobs-part-1/
Section 5 of the original Spark paper published on NSDI'12 by Zaharia, Matei, et al

Spark Driver and Executor



Driver program

```
sc = SparkContext(appName="Average Rating per Genre")

#You can change the input path pointing to your own HDFS
#If spark is able to read hadoop configuration, you can use relative path
input_path = 'hdfs://soit-hdp-pro-1.ucc.usyd.edu.au/share/movie/small/'

#Relative path is used to specify the output directory
#The relative path is always relative to your home directory in HDFS: /user/<yourUserName>
output_path = 'ratingOut'

ratings = sc.textFile(input_path + "ratings.csv")
movieData = sc.textFile(input_path + "movies.csv")

movieRatings = ratings.map(extractRating)
movieGenre = movieData.flatMap(pairMovieToGenre) # we use flatMap as there are multiple genre

genreRatings = movieGenre.join(movieRatings).values()
genreRatingsAverage = genreRatings.aggregateByKey((0.0,0),
                                                  mergeRating,
                                                  mergeCombiners, 1).map(mapAverageRating)

genreRatingsAverage.saveAsTextFile(output_path)
```

Runs in an executor, maybe on a different machine

Runs in an executor, maybe on a different machine s

Runs in an executor, maybe on a different machine

Runs in an executor, maybe on a different machine

Runs in an executor, maybe on a different machine

Runs in an executor, maybe on a different machine

Driver collects back the results and save it in a file

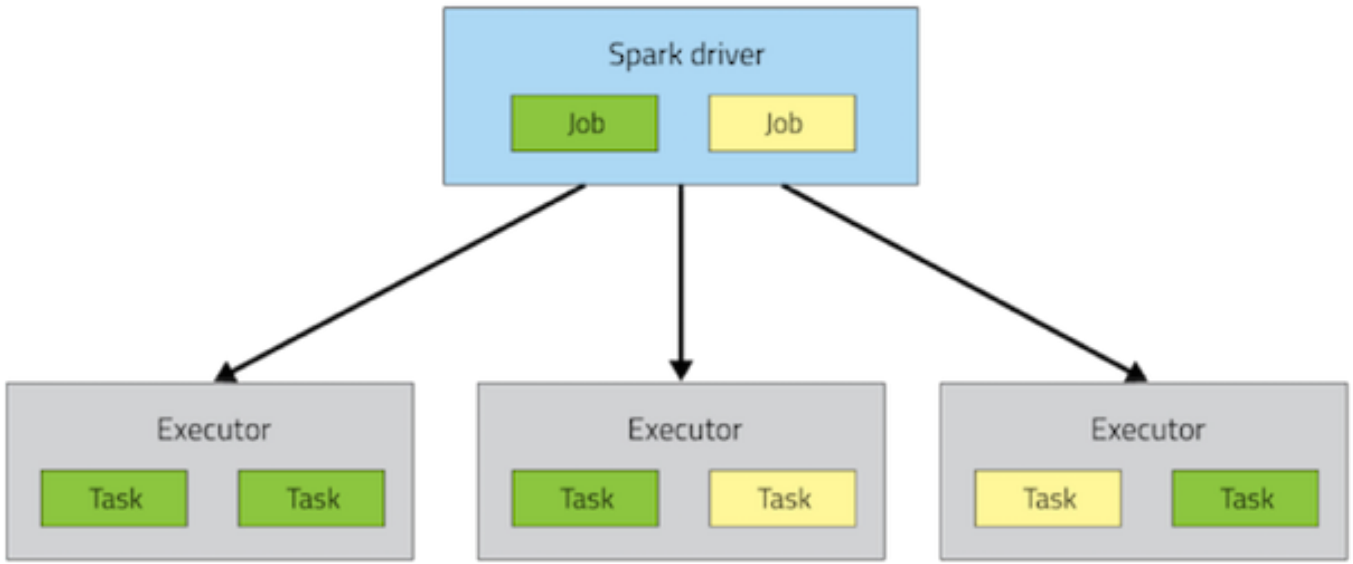
Spark Lazy Evaluation Principle

- All transformations in Spark are *lazy*, in that they do not compute their results right away. Instead, they just remember the transformations applied to some base dataset (e.g. a file). The transformations are only computed when an action requires a **result** to be returned to the driver program.
  - When you call **count()** on an RDD, all transformations leading to the construction of this RDD will be executed.
  - This includes the very first file reading operation
- Benefits: Better optimization by analysing the whole sequence of transformations
  - Transformations are usually grouped together as stages (late slides) to avoid unnecessary data shuffling

Spark Lazy Evaluation Consequence

- Because each action has its own data flow graph (lineage graph), there are possible unnecessary re-computations
  - What if I use **`count()`** for each intermediate RDDs
    - This will cause a lot of re-computations indeed!
    - Using `count()`, `take()`, `first()`, … for debugging purpose is appropriate
- If the computation branches out, e.g. a common RDD is used in two or more different sequences of of transformations, each ended with an action.
  - Call **`cache()`** or **`persist()`** on the common RDD to keep it in the memory or other storage level
  - **`cache()`** is a special case of **`persist()`**, which keeps the RDD in the memory
  - **`persist()`** accepts argument for different storage levels, but the default one is to keep RDD in the memory

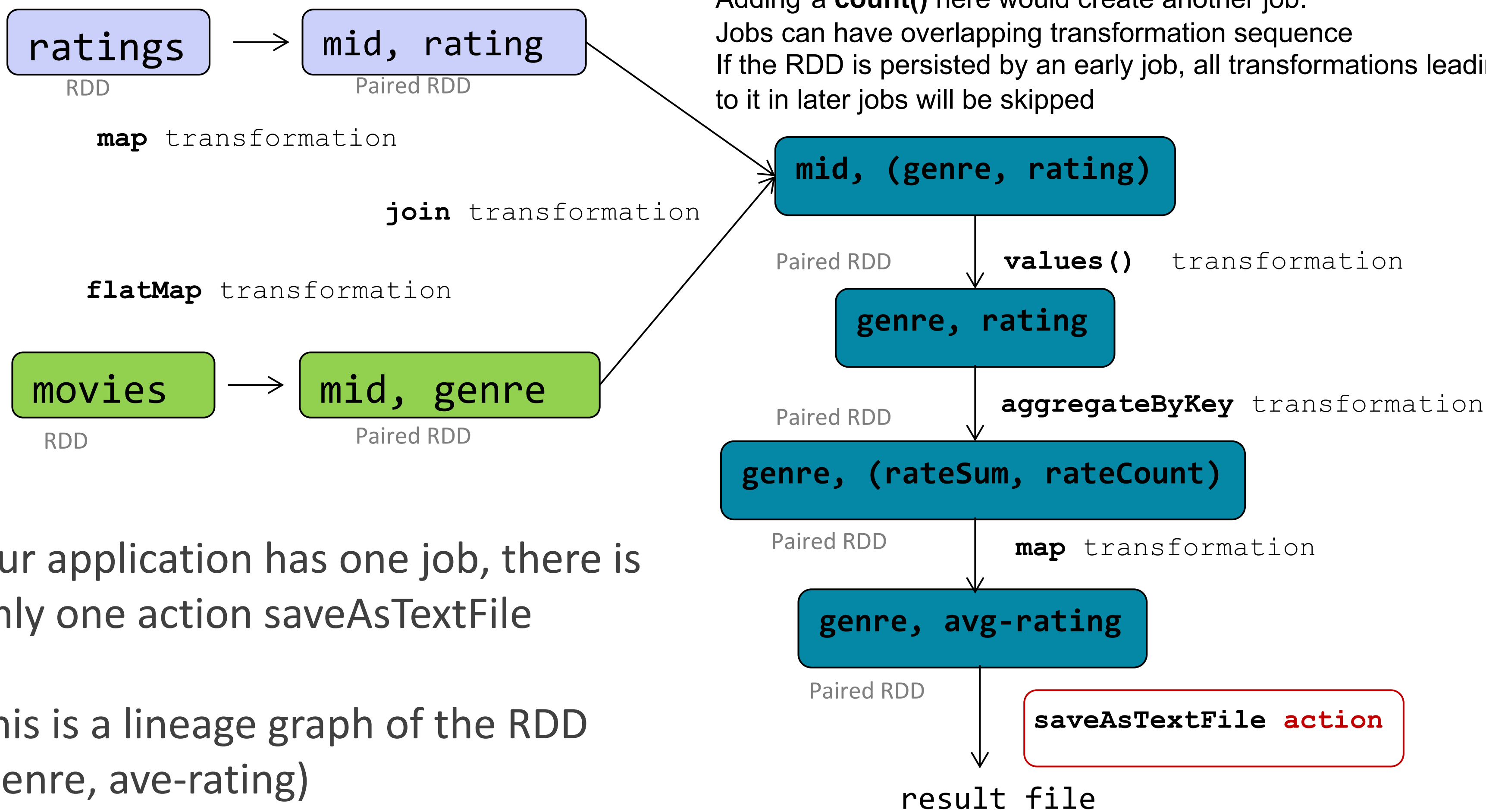
Spark Application in Execution



Job, Stage and Task

- Job is triggered by actions such as **`count`**, **`save`**, etc

Adding a **count()** here would create another job.
Jobs can have overlapping transformation sequence
If the RDD is persisted by an early job, all transformations leading to it in later jobs will be skipped



Our application has one job, there is only one action saveAsTextFile

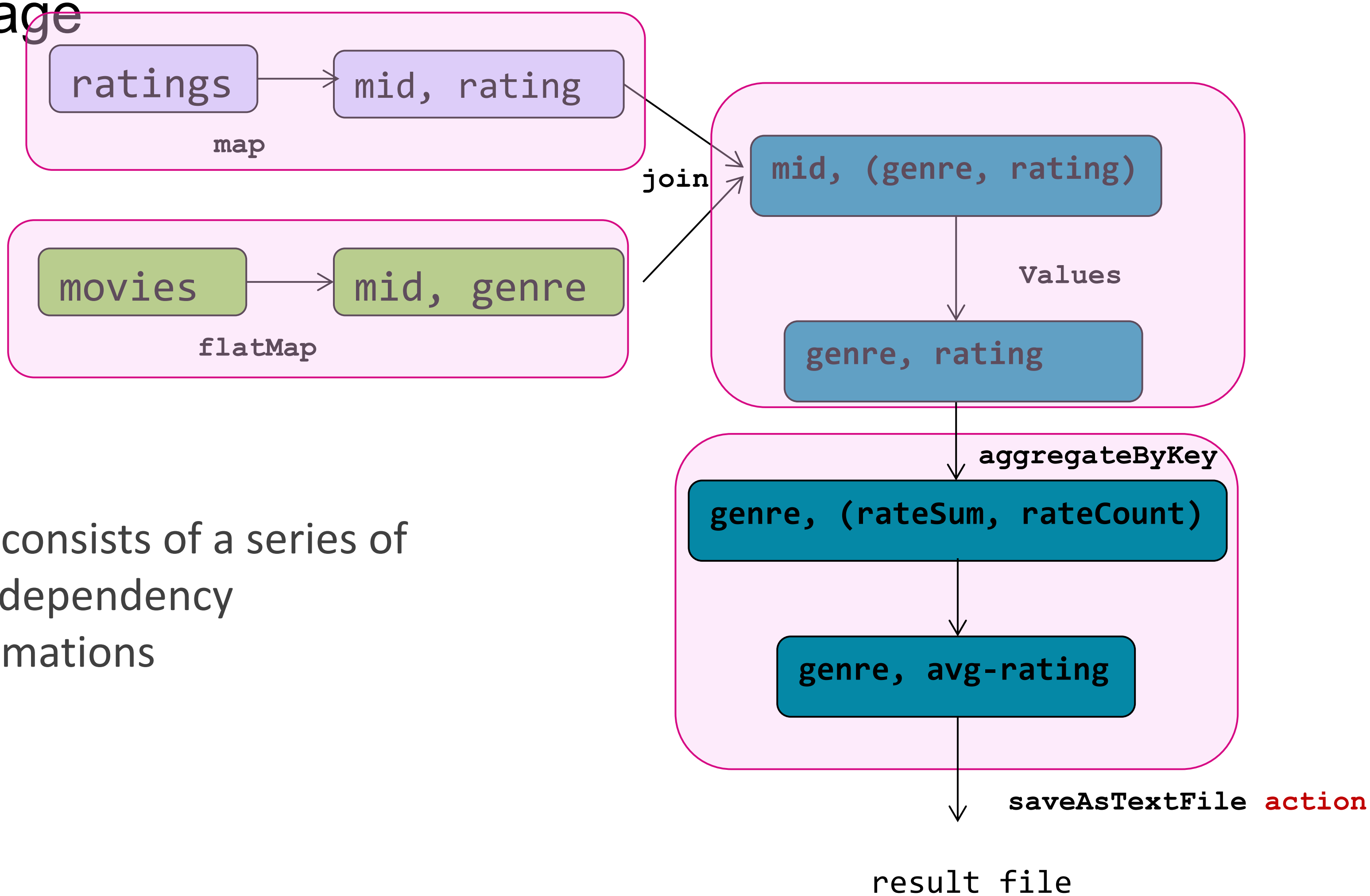
This is a lineage graph of the RDD (genre, ave-rating)

Narrow and Wide Dependencies

- A transformation can cause *narrow* or *wide* dependencies between the parent and child RDD
- **Narrow dependency** means one partition of the child RDD can be computed by only one or limited partition of the parent RDD
  - All map like transformations (map, flatMap, filter, etc) have narrow dependency
- **Wide dependency** means one partition of the child RDD may need data from all partitions of the parent RDD
  - All reduce like transformations (reduceByKey, groupByKey, join, etc) have wide dependency in general
  - In certain circumstance, they may have narrow dependency (see later slides).

Job, Stage and Task

- A job can have many stages as a DAG based on the RDD's lineage

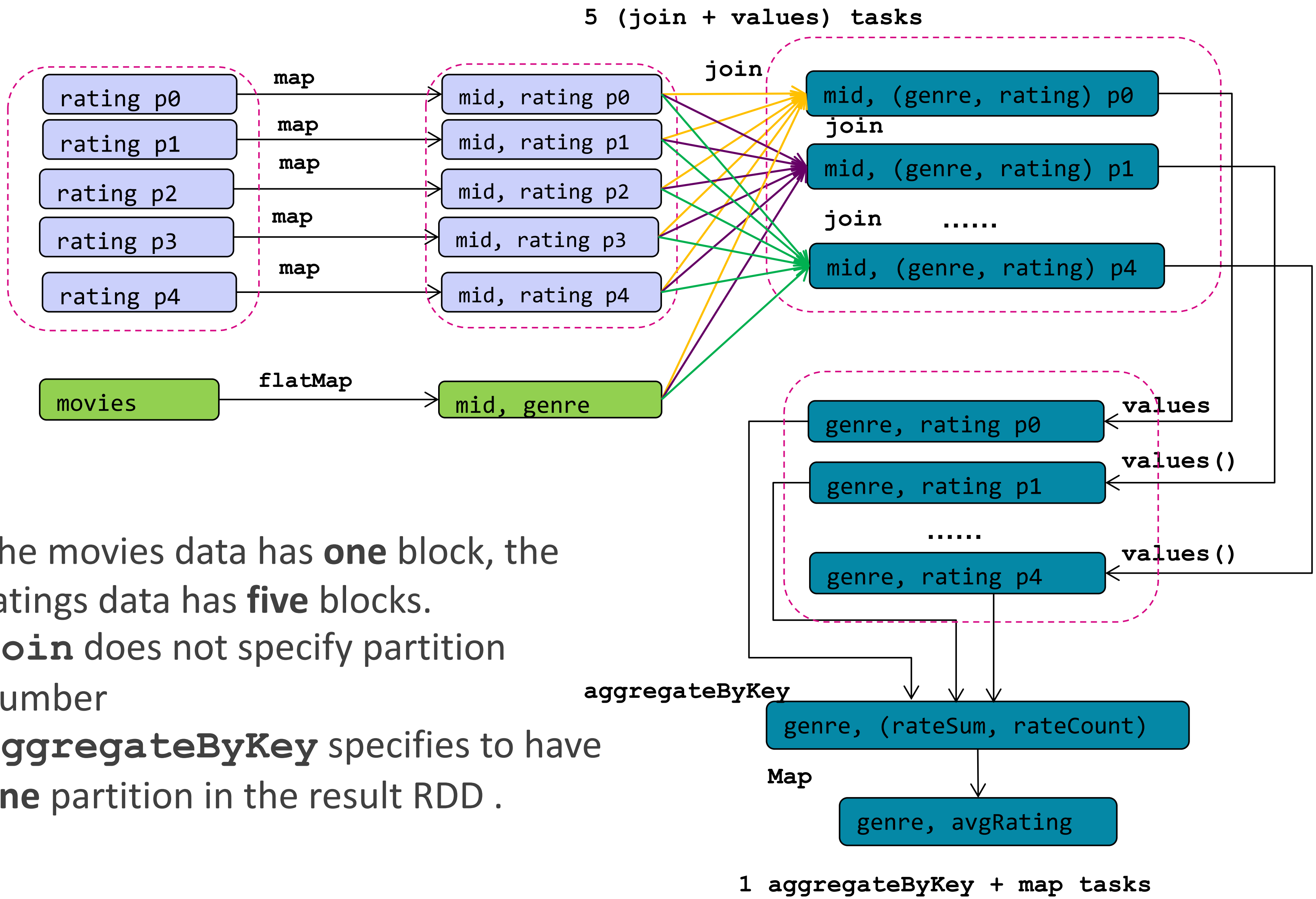


A **stage** consists of a series of narrow dependency transformations

Job, Stage and Task

- Transformations inside a stage can execute in a pipeline style to improve efficiency
  - There is no global data shuffle inside a stage
- There will be data shuffling across stages
  - Similar to shuffle in MapReduce framework
- The pipelined transformations inside a stage represent a task
  - Task is the schedulable execution unit of an application
    - Analogy to the map and reduce task in MapReduce
  - The number of tasks of an application depends on the number of partitions the parent RDD has
  - Wide dependency transformation can take a number of partition parameter

Job, Stage and Task



The movies data has **one** block, the ratings data has **five** blocks.

`join` does not specify partition number

`aggregateByKey` specifies to have **one** partition in the result RDD .

Comparison between MR and Spark

- MapReduce *Job*:
  - Consists of M Mappers (map tasks) and/or R Reducers(reduce tasks)
- Spark *job* is not the equivalent of MapReduce job
  - It could be smaller than or larger than MapReduce job
  - The previous spark application would needs two MapReduce jobs
- Spark *stage* is comparable with MapReduce's map/reduce phase, in particular, with respect to shuffling
- Spark *task* is comparable with MapReduce's task
- Spark executors stick around throughout the application execution time
  - Each typically run multiple tasks of different type
    - E.g. the **join** task might run in the same executor of the **map** task
  - Much better data sharing within executor

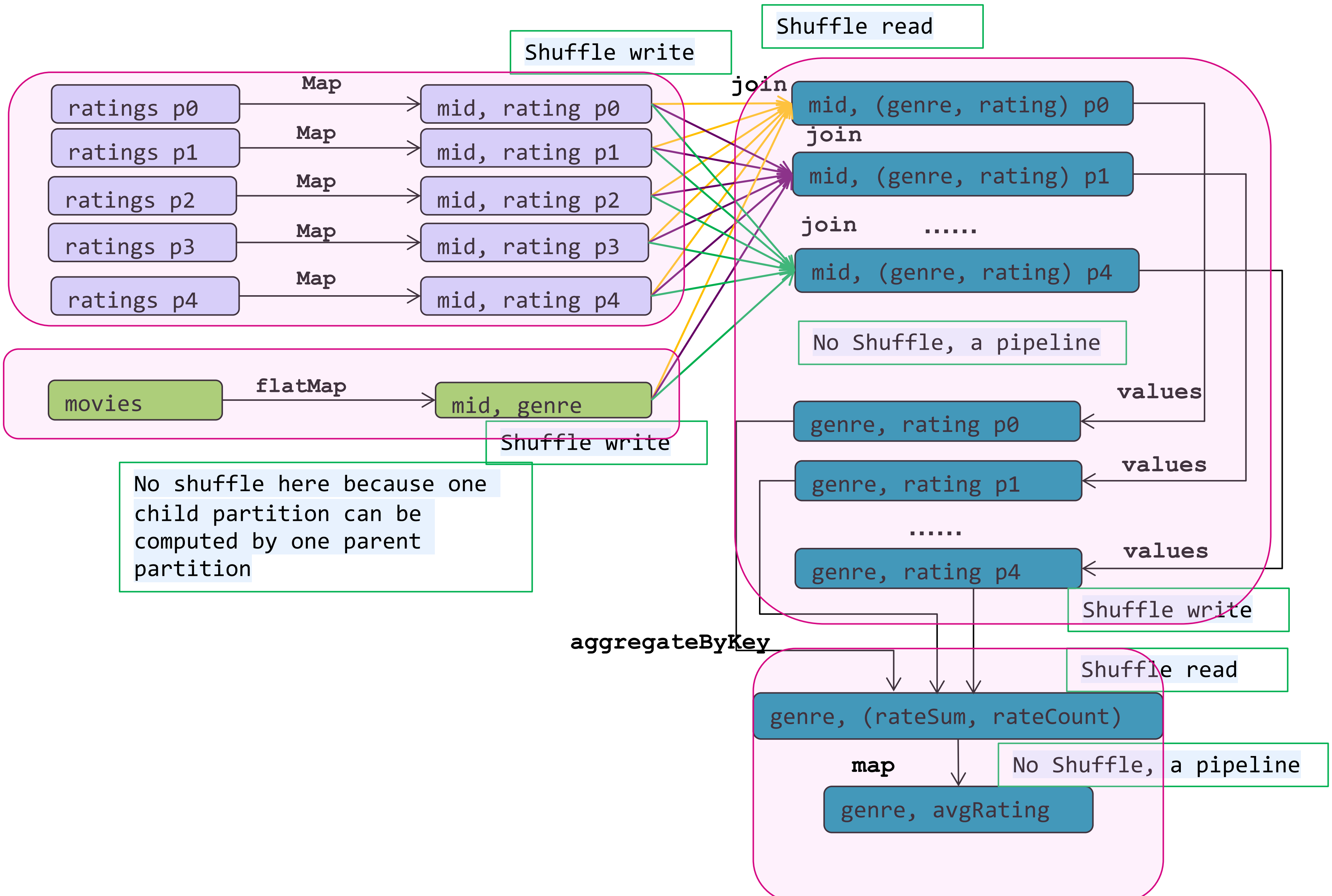
Outline

- Spark Execution View
- **Impact of Shuffling**
- Cluster Resource Specification

Shuffle and its impact

- Shuffles are fairly expensive; all shuffle data must be **written to disk** and then transferred over the network.
- Design and choose your transformations carefully to avoid shuffling too much data
- Transformations causing shuffle also stress memory if not designed properly
  - Any **`join`**, **`*ByKey`** operation involves holding objects in hashmaps or in-memory buffers to group or sort.
  - It is preferred to have more small tasks to reduce memory stress in individual node

When shuffles happen



When shuffles do not happen

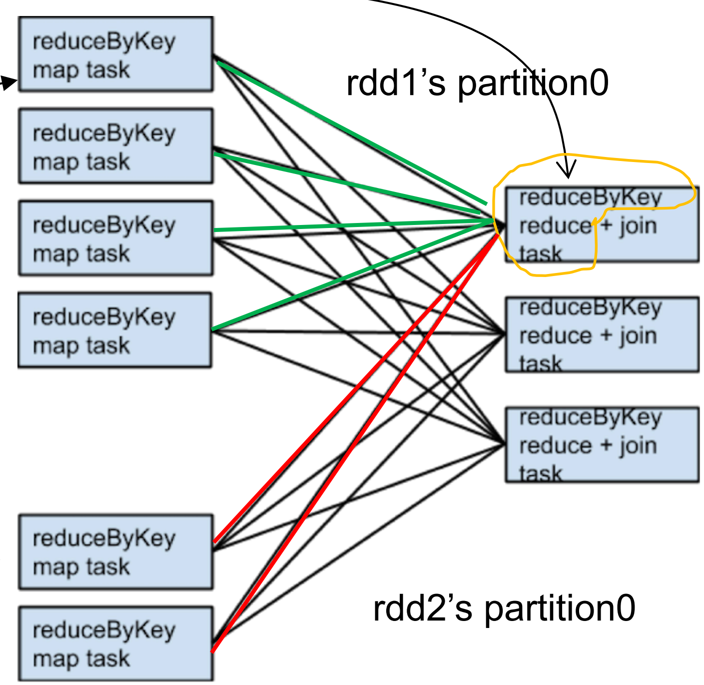
- There are cases where **`join`** or **`*ByKey`** operation does not involve shuffle and would not trigger stage boundary
  - If child and parent RDDs are partitioned by the same partitioner and/or have the same number of partition

Join with input co-partitioned

3 partitions 4 partitions
```
rdd1 = someRdd.reduceByKey(...)
```
3 partitions 2 partitions
```
rdd2 = someOtherRdd.reduceByKey(...)
rdd3 = rdd1.join(rdd2)
```
3 partitions 3 partitions 3 partitions

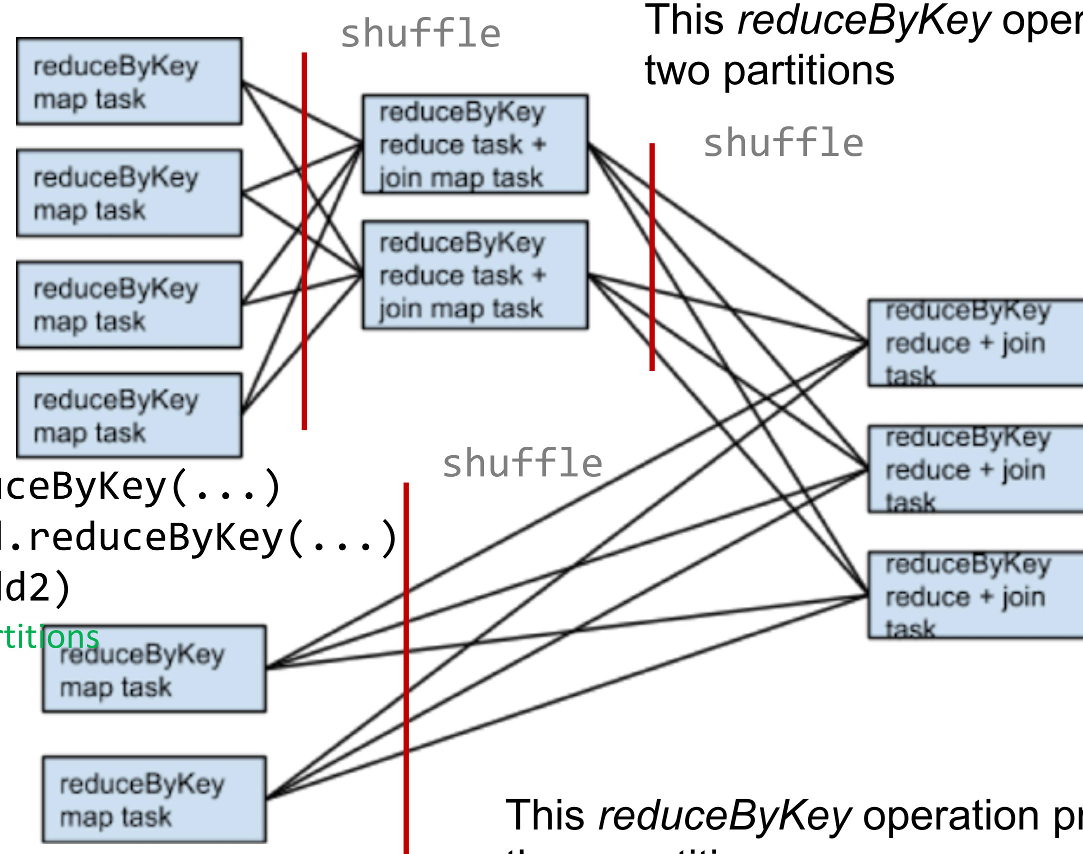
The two *reduceByKey* operations use the same key and end up with 3 partitions.

rdd1's partition0 and rdd2's partition 0 contain the same set of keys and can join to produce rdd3's partition0, this is narrow dependency



When shuffles do not happen (cont’d)

- If the parent RDD uses the same partitioner, but result in different number of partitions. Only one parent RDD needs to be re-shuffled for the join operation



This *reduceByKey* operation produces two partitions

The join operation, by default, adopts the larger partition number from its partens.

This *reduceByKey* operation produces three partitions

2 partitions 4 partitions

```
rdd1 = someRdd.reduceByKey(...)
```

3 partitions 2 partitions

```
rdd2 = someOtherRdd.reduceByKey(...)
rdd3 = rdd1.join(rdd2)
```

2 partitions 3 partitions

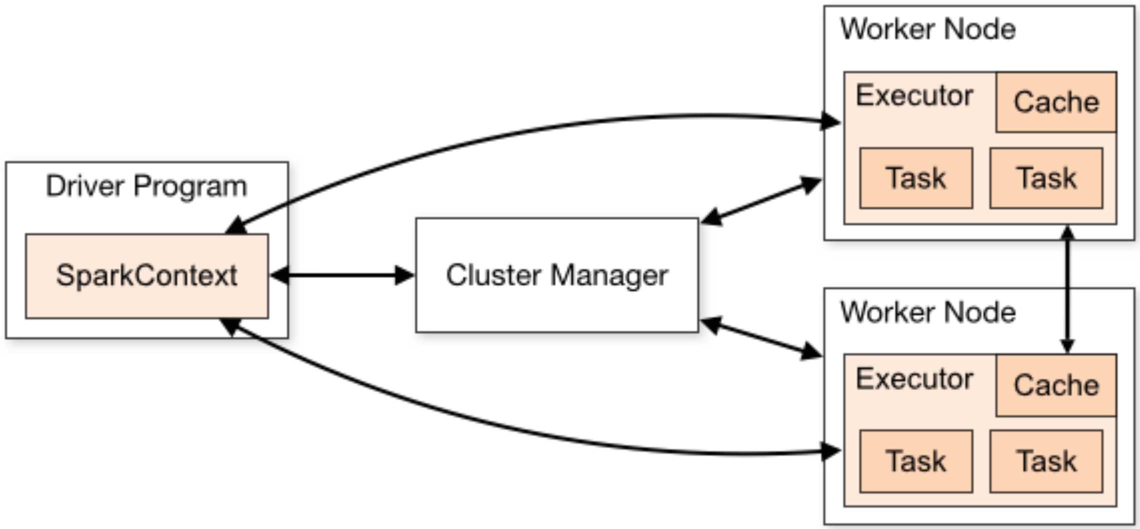
3 partitions

Outline

- **Spark Execution View**
- **Impact of Shuffling**
- **Cluster Resource Specification**
  - Specifying Available Resources for YARN
  - Specifying default resource requirements for Spark application
  - Specifying individual application resource requirements

Cluster Deployment Modes

YARN/Mesos/Standalone



Depending on where the driver is running, spark application can be submitted in either *cluster* or *client* mode

With YARN cluster mode, the driver runs in a container, which also runs the **AM** and each executor runs in a container

With YARN and client, the driver runs on separate process, AM runs in a container, each executor runs in a container

How to submit Spark Application

- Spark applications are usually submitted using **spark-submit** script
  - Specify a few important parameters
- For debugging on local installation, you can set all important parameters in a configuration object and pass it to **SparkContext.** The program then run as regular Python application.

```
spark-submit \
    --master yarn \
    --deploy-mode cluster \
    --num-executors 3 \
    --py-files ml_utils.py AverageRatingPerGenre.py \
    --input movies/ \
    --output genre-avg-python/
```

System Wide Resource Specification

- Main resource types
  - Memory size
  - Core number
    - Control the level of parallelism, one thread per core
- YARN usually knows system wide available resources
  - Total memory per host
    - **`yarn.nodemanager.resource.memory-mb`**
  - Total core per host
    - **yarn.nodemanager.resource.cpu-vcores**
- These values were set in a configuration file
  - In EMR: `/etc/hadoop/conf/yarn-site.xml`

Sample System Resources

3 node Hadoop Cluster with m4.xlarge instance, each with 8 vCPU, 32G memory

```
<property>
  <name>yarn.nodemanager.resource.cpu-vcores</name>
  <value>8</value>
</property>
<property>
  <name>yarn.nodemanager.resource.memory-mb</name>
  <value>12288</value>
</property>
```

Cluster Metrics

Apps Submitted	Apps Pending	Apps Running	Apps Completed	Containers Running	Memory Used	Memory Total	Memory Reserved	VCores Used	VCores Total
0	0	0	0	0	0 B	24 GB	0 B	0	16

Cluster Nodes Metrics

Active Nodes	Decommissioning Nodes	Decommissioned Nodes	Lost Nodes	Unhealthy Nodes	Rebooted Nodes	S
2	0	0	0	0	0	0

Spark Application Resource Specification

- Driver resource specification
  - **`spark.driver.memory`**
    - Amount of memory to use for the driver process
    - Default: 1G
  - **`spark.driver.cores`**
    - Number of cores to use for the driver process, only in cluster mode.
    - Default:1
- Executor resource specification
  - **`spark.executor.memory`**
    - Amount of memory to use per executor process
    - Default: 1G
  - **`spark.executor.cores`**
    - The number of cores to use on each executor
    - Default 1
  - **`spark.default.parallelism`**
    - Default number of partitions in RDDs returned by transformations like join, reduceByKey,
    - Default: 20

Spark Application Resource Specification

- Many properties have default values set in configuration file
  - In EMR: `/etc/spark/conf/spark-defaults.conf`

```
spark.executor.memory                9486M
spark.executor.cores                 4
spark.driver.memory                  2048M
spark.dynamicAllocation.enabled      true
```

~ 9.26G, I made a mistake in the recording

- Per application setting can be specified on the spark-submit script

spark-submit \

--master yarn \

--deploy-mode cluster \

-- *num-executors 3* \

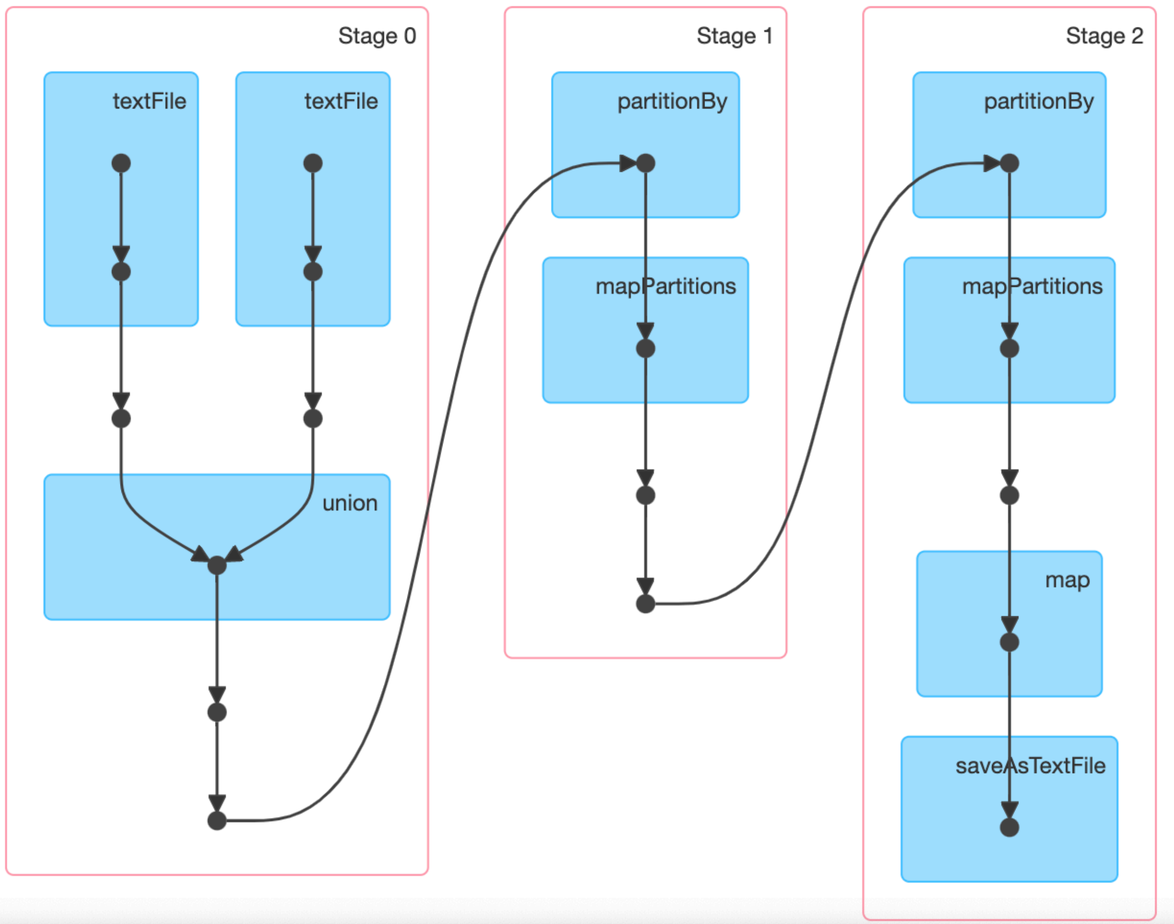
-- executor-cores 8 \

-- executor-memory 12G

Application History Screen Shot

Completed Jobs (1)

Job Id	Description	Submitted	Duration	Stages: Succeeded/Total	Tasks (for all stages): Succeeded/Total
0	runJob at SparkHadoopWriter.scala:78 runJob at SparkHadoopWriter.scala:78	2020/04/07 06:58:59	2.1 min	3/3	15/15



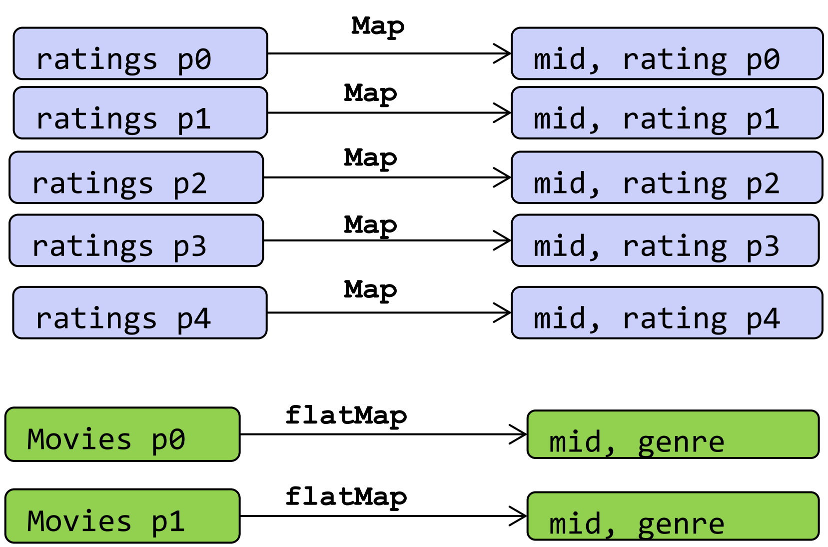
Application History Screenshot: stages

Completed Stages: 3

Completed Stages (3)

Stage Id	Description	Submitted	Duration	Tasks: Succeeded/Total	Input	Output	Shuffle Read	Shuffle Write
2	runJob at SparkHadoopWriter.scala:78 +details	2020/04/07 07:01:02	0.2 s	1/1		633.0 B	4.5 KB	
1	aggregateByKey at AverageRatingPerGenre.py:29 +details	2020/04/07 07:00:19	43 s	7/7			134.6 MB	4.5 KB
0	join at AverageRatingPerGenre.py:28 +details	2020/04/07 06:58:59	1.3 min	7/7	543.9 MB			134.6 MB

Spark starts two tasks for file with only one block



Block information -- Block 0

Block ID: 1073741825

Block Pool ID: BP-832592447-172.31.70.14-1586237005251

Generation Stamp: 1001

Size: 134217728

Block information -- Block 4

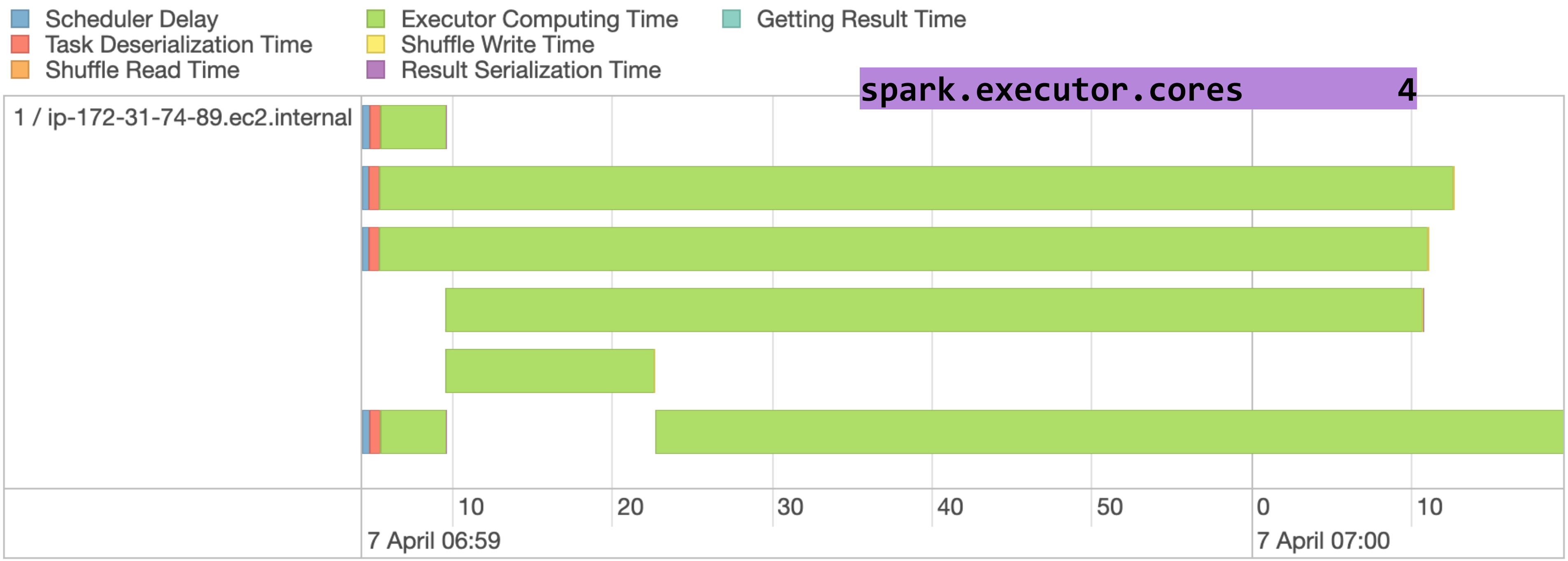
Block ID: 1073741829

Block Pool ID: BP-832592447-172.31.70.14-1586237005251

Generation Stamp: 1005

Size: 31416403

Stage Task Execution Details



At any time point, 4 tasks are running in parallel in one executor , because the executor is set to have 4 cores

5 tasks running on ratings data (542M in 5 blocks)
2 tasks running on movies data (1.7M in one block, with 2 splits)

Executor View

Executor ID	Address	Status	RDD Blocks	Storage Memory	Disk Used	Cores	Active Tasks	Failed Tasks	Complete Tasks
driver	ip-172-31-74-152.ec2.internal:38997	Active	0	0.0 B / 1.1 GB	0.0 B	0	0	0	0
1	ip-172-31-74-89.ec2.internal:37965	Active	0	0.0 B / 5.8 GB	0.0 B	4	0	0	15

```
spark-submit \
  --master yarn \
  --deploy-mode cluster \
  --num-executors 3 \
  --py-files ml_utils.py AverageRatingPerGenre.py \
  --input movies/ \
  --output genre-avg-python/
```

```
spark.dynamicAllocation.enabled  true
```

```
spark.executor.memory                       9486M
spark.driver.memory                         2048M

spark.yarn.executor.memoryOverheadFactor 0.1875
```

Another Execution Plan

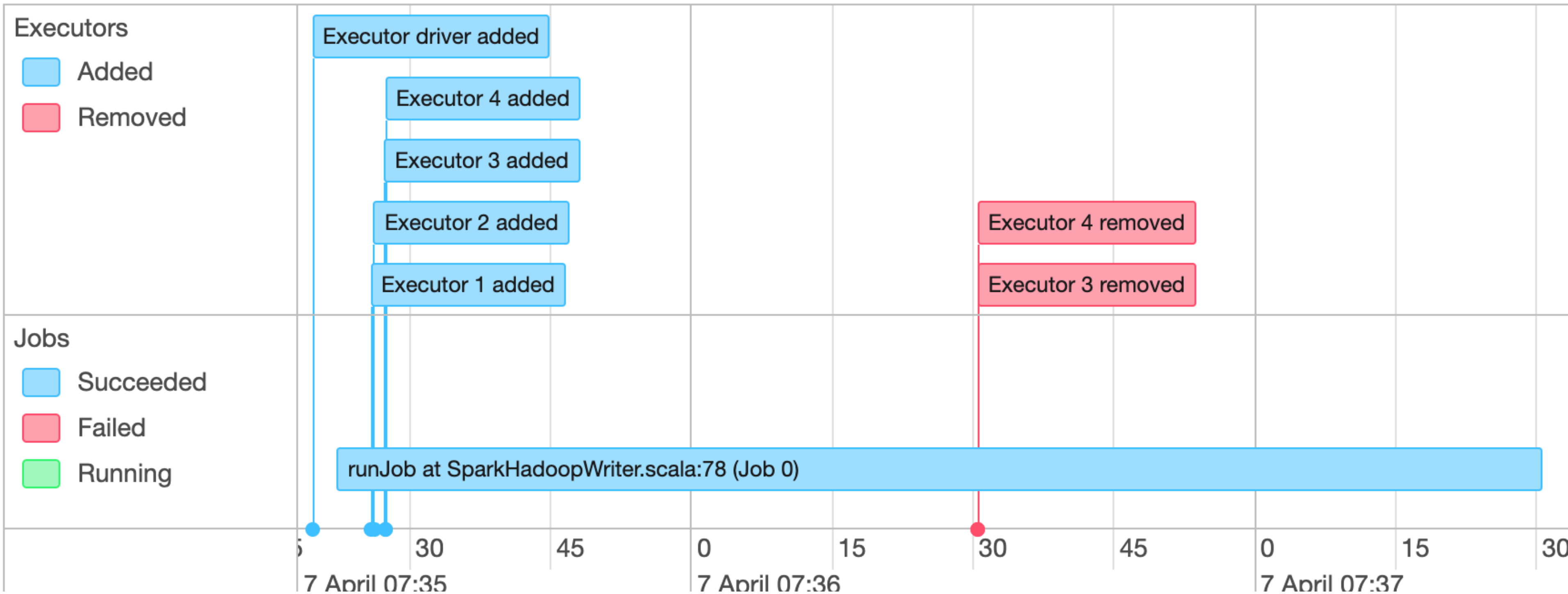
```
spark-submit \
    --master yarn \
    --deploy-mode cluster \
    --executor-memory 4G \
    --py-files ml_utils.py AverageRatingPerGenre.py \
    --input movies/ \
    --output output_2/
```

Executors

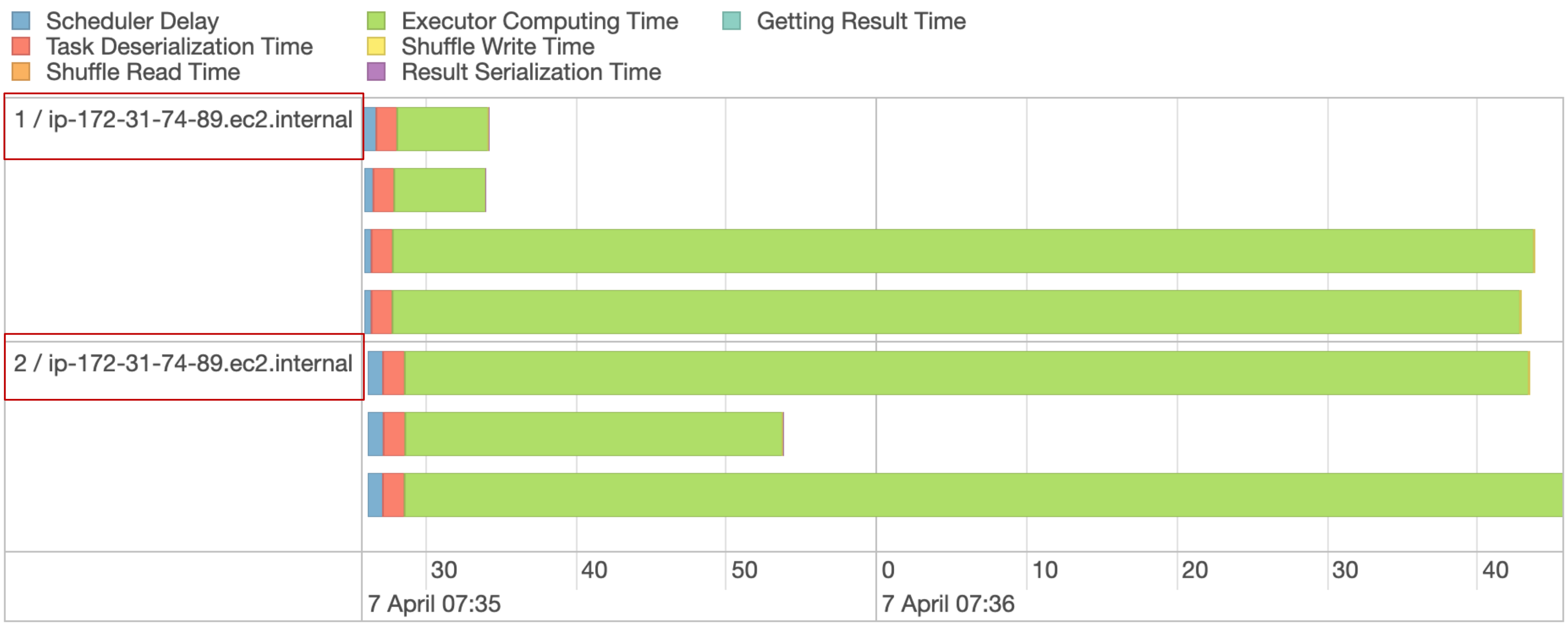
Show 20 entries

Executor ID	Address	Status	RDD Blocks	Storage Memory	Disk Used	Cores	Active Tasks	Failed Tasks	Complete Tasks	Total Tasks	Task Time (GC Time)
driver	ip-172-31-74-89.ec2.internal:44417	Active	0	0.0 B / 1.1 GB	0.0 B	0	0	0	0	0	0 ms (0 ms)
1	ip-172-31-74-89.ec2.internal:33281	Active	0	0.0 B / 2.4 GB	0.0 B	4	0	0	9	9	5.7 min (5 s)
2	ip-172-31-74-89.ec2.internal:34407	Active	0	0.0 B / 2.4 GB	0.0 B	4	0	0	6	6	5.2 min (4 s)
3	ip-172-31-74-152.ec2.internal:37373	Dead	0	0.0 B / 2.4 GB	0.0 B	4	0	0	0	0	0 ms (0 ms)
4	ip-172-31-74-152.ec2.internal:35085	Dead	0	0.0 B / 2.4 GB	0.0 B	4	0	0	0	0	0 ms (0 ms)

Executors Requested Then Removed



Stage 0 Task Execution in Two Executors



2 executors on the same node

Data Locality

- Data locality means how close data is to the code processing it
  - Both MapReduce and Spark work on the principle of shipping code instead of data
- Spark Data Locality Levels
  - PROCESS_LOCAL data is in the same JVM as the running code. This is the best locality possible
  - NODE_LOCAL data is on the same node. Examples might be in HDFS on the same node, or in another executor on the same node. This is a little slower than PROCESS_LOCAL because the data has to travel between processes
  - NO_PREF data is accessed equally quickly from anywhere and has no locality preference
  - RACK_LOCAL data is on the same rack of servers. Data is on a different server on the same rack so needs to be sent over the network, typically through a single switch
  - ANY data is elsewhere on the network and not in the same rack

Data Locality Information

▾ Tasks (7)

Index ▲	ID	Attempt	Status	Locality Level	Executor ID	Host	Launch Time	Duration	GC Time	Input Size / Records	Write Time	Shuffle Write Size / Records
0	0	0	SUCCESS	NODE_LOCAL	1	ip-172-31-74-89.ec2.internal stdout stderr	2020/04/07 07:35:25	6 s	0.4 s	896.0 KB / 17052	80 ms	639.0 KB / 84
1	1	0	SUCCESS	NODE_LOCAL	1	ip-172-31-74-89.ec2.internal stdout stderr	2020/04/07 07:35:25	6 s	0.4 s	844.6 KB / 17156	24 ms	597.1 KB / 84
2	2	0	SUCCESS	NODE_LOCAL	1	ip-172-31-74-89.ec2.internal stdout stderr	2020/04/07 07:35:25	1.3 min	0.9 s	128.1 MB / 5111286	0.1 s	32.3 MB / 112
3	3	0	SUCCESS	NODE_LOCAL	1	ip-172-31-74-89.ec2.internal stdout stderr	2020/04/07 07:35:25	1.3 min	0.9 s	128.1 MB / 5036778	99 ms	31.8 MB / 105
4	6	0	SUCCESS	RACK_LOCAL	2	ip-172-31-74-89.ec2.internal stdout stderr	2020/04/07 07:35:26	1.3 min	0.9 s	128.1 MB / 4885562	72 ms	30.9 MB / 112
5	4	0	SUCCESS	NODE_LOCAL	2	ip-172-31-74-89.ec2.internal stdout stderr	2020/04/07 07:35:26	1.2 min	0.9 s	128.1 MB / 4885596	0.1 s	30.9 MB / 112
6	5	0	SUCCESS	NODE_LOCAL	2	ip-172-31-74-89.ec2.internal	2020/04/07 07:35:26	25 s	0.6 s	30.0 MB / 1143906	91 ms	7.4 MB / 84

References

- Spark Documentation

- Sandy Ryza, How-to: Tune Your Apache Spark Jobs (part 1) published on March 09, 2013
  - http://blog.cloudera.com/blog/2015/03/how-to-tune-your-apache-spark-jobs-part-1/
- Sandy Ryza, How-to: Tune Your Apache Spark Jobs (part 2) published on March 30, 201
  - http://blog.cloudera.com/blog/2015/03/how-to-tune-your-apache-spark-jobs-part-2/