THE DATA SCIENCE LAB
Advanced Spark Optimization

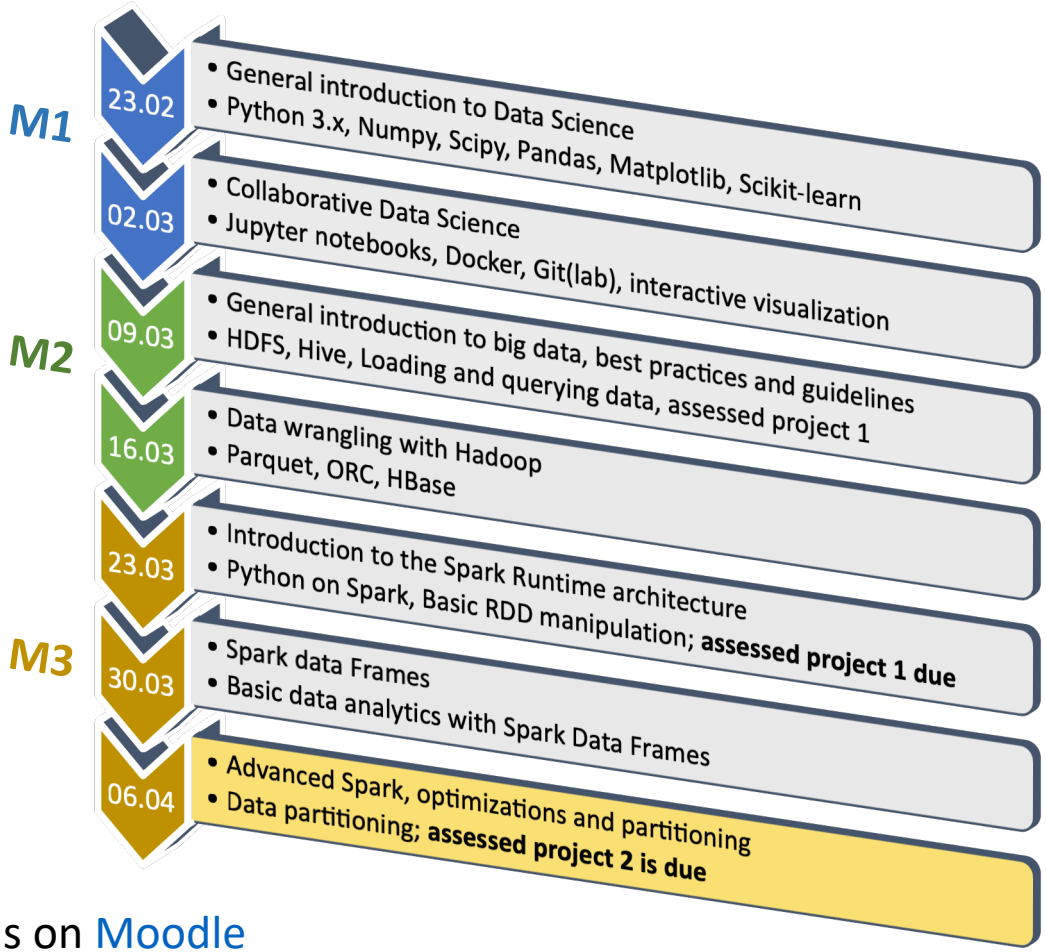
COM 490 – Spring 2022

Week 7

THIS CLASS WILL BE RECORDED

EPFL

Agenda Spring 2022



*Details on Moodle

Week 6 Recap

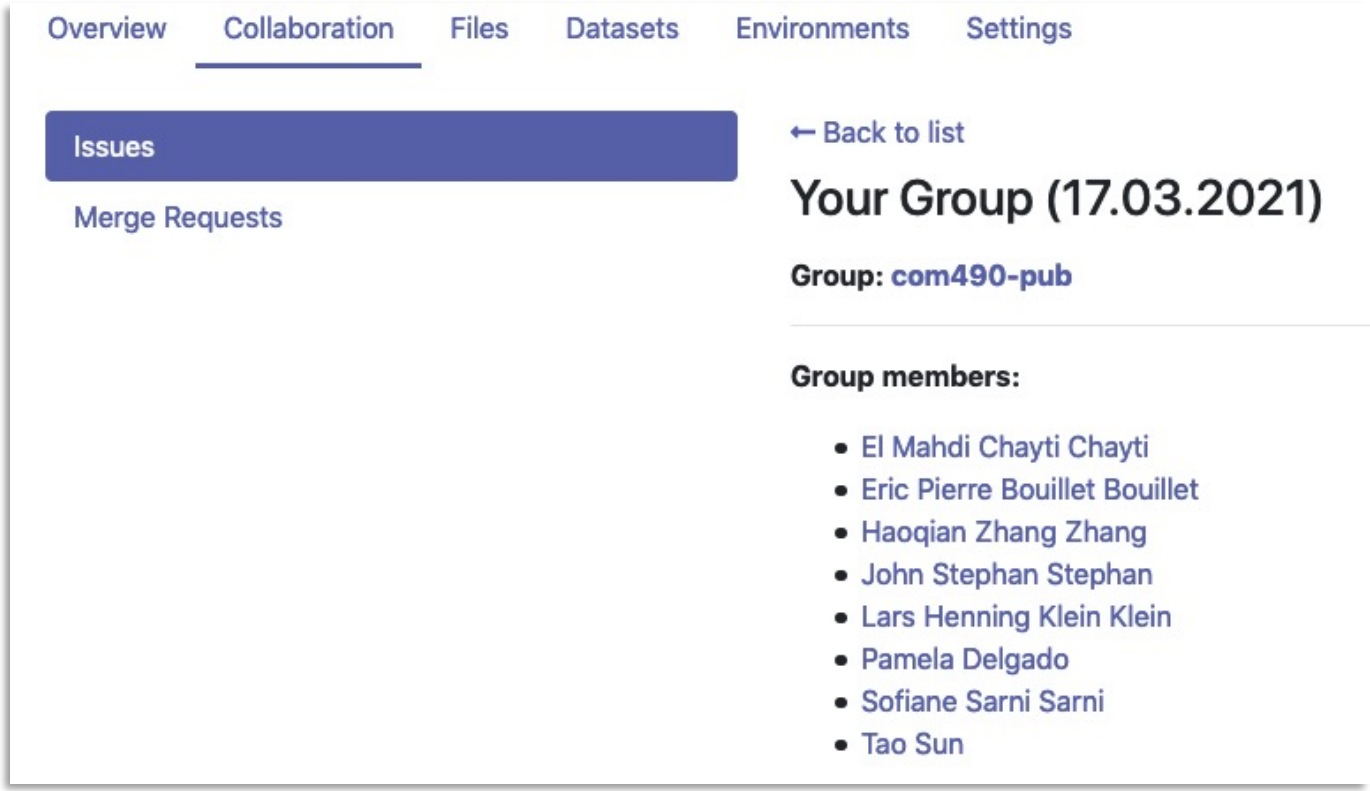
- Revisit RDDs
- DataFrames
  - Structured data
  - Higher level API, declarative programming
- RDDs vs DataFrames
- DataFrames demo
- DataFrames under the hood
  - PySpark internals (Py4J)
  - Catalyzer optimization (query plan optimization)
- Exercises: Guttenberg pt2

Week 6 – Questions?

Week 7 – Homework 2 : Checkpoint

Have you seen this in your homework project ?

If not, contact us !



Today's Agenda

- Spark parallelization
- Spark and Big Data
- The cost of shuffle operations and memory
- Spark partitioning
- Partitioning demo
- Exercise:
  - Spark DataFrames with partitioning
- Homework 3

Introduction to Spark Parallelization

RDD - Revisited

- RDD - Resilient Distributed Dataset
  - Immutable: once an RDD is created it cannot be modified, actions create new RDDs
  - Lineage: RDD points to its parents
  - Resilient: if a node dies data is easily regenerated
  - Partitioned → pieces of RDD ***distributed*** over the cluster

RDD – parallelize API

- Creation of RDD

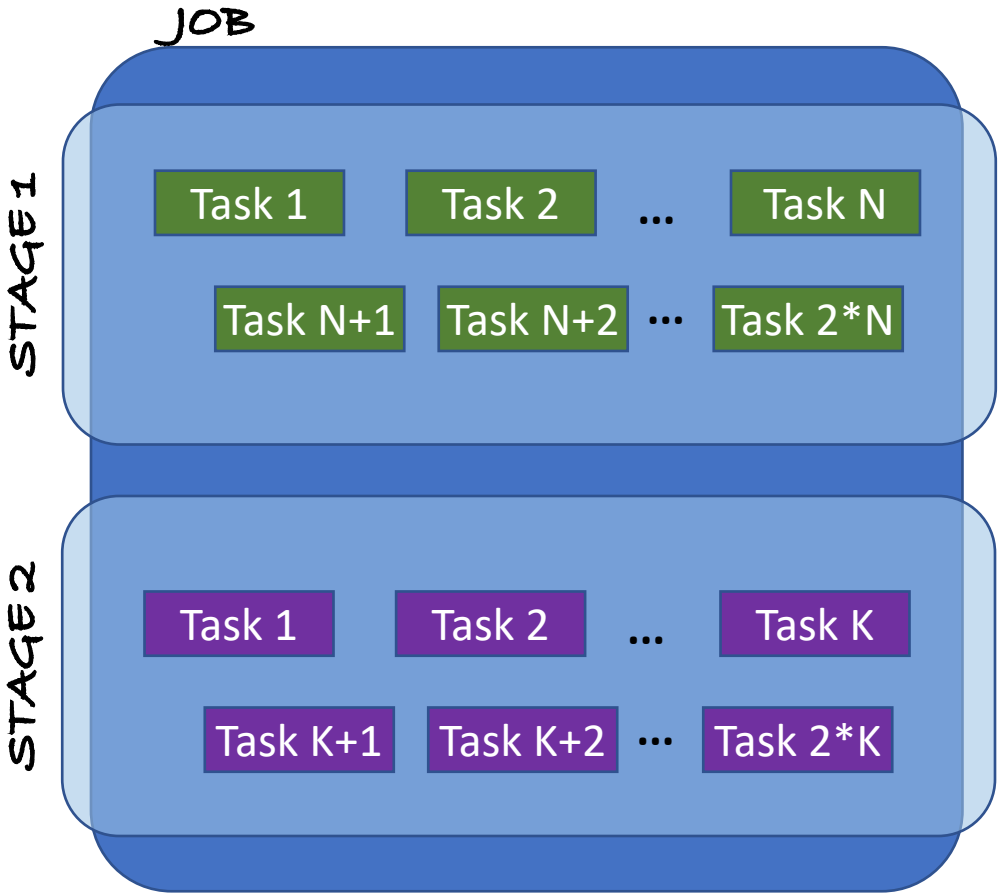
```
sc.parallelize(data)
```

  - Reference external or local data
  - Each piece of data becomes a partition
  - Each partition is processed in a task
  - Tasks are executed in parallel

- Why parallelize?
  - Take advantage of distributed resources

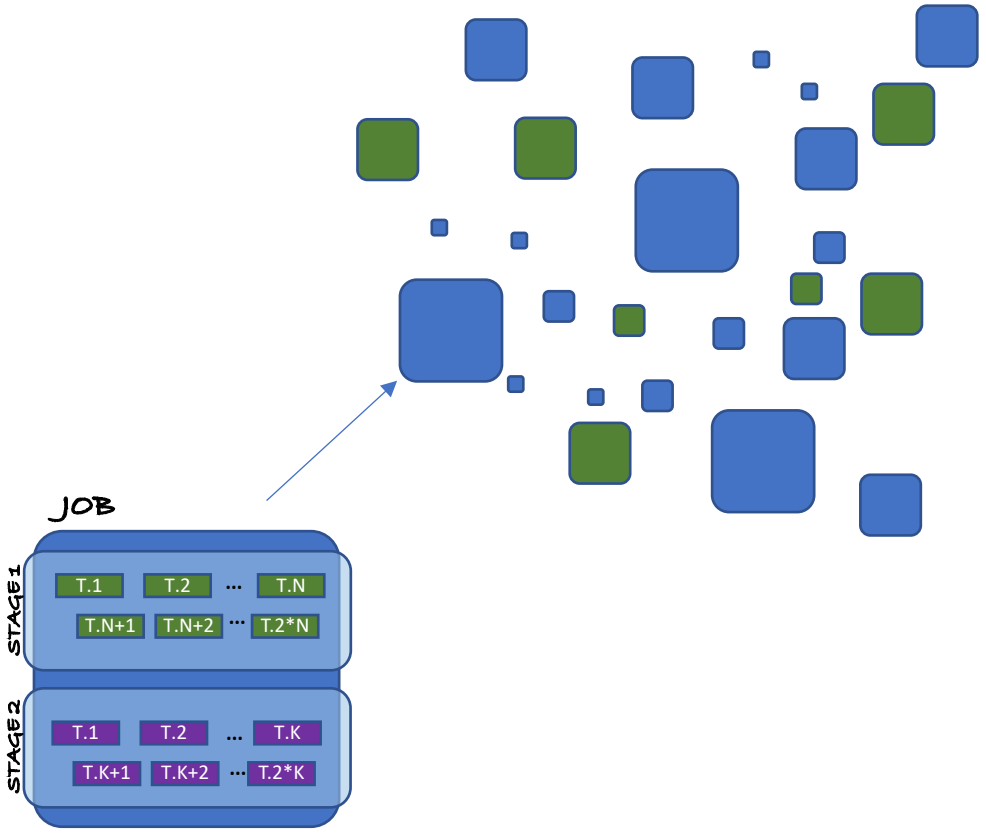
Spark units of work

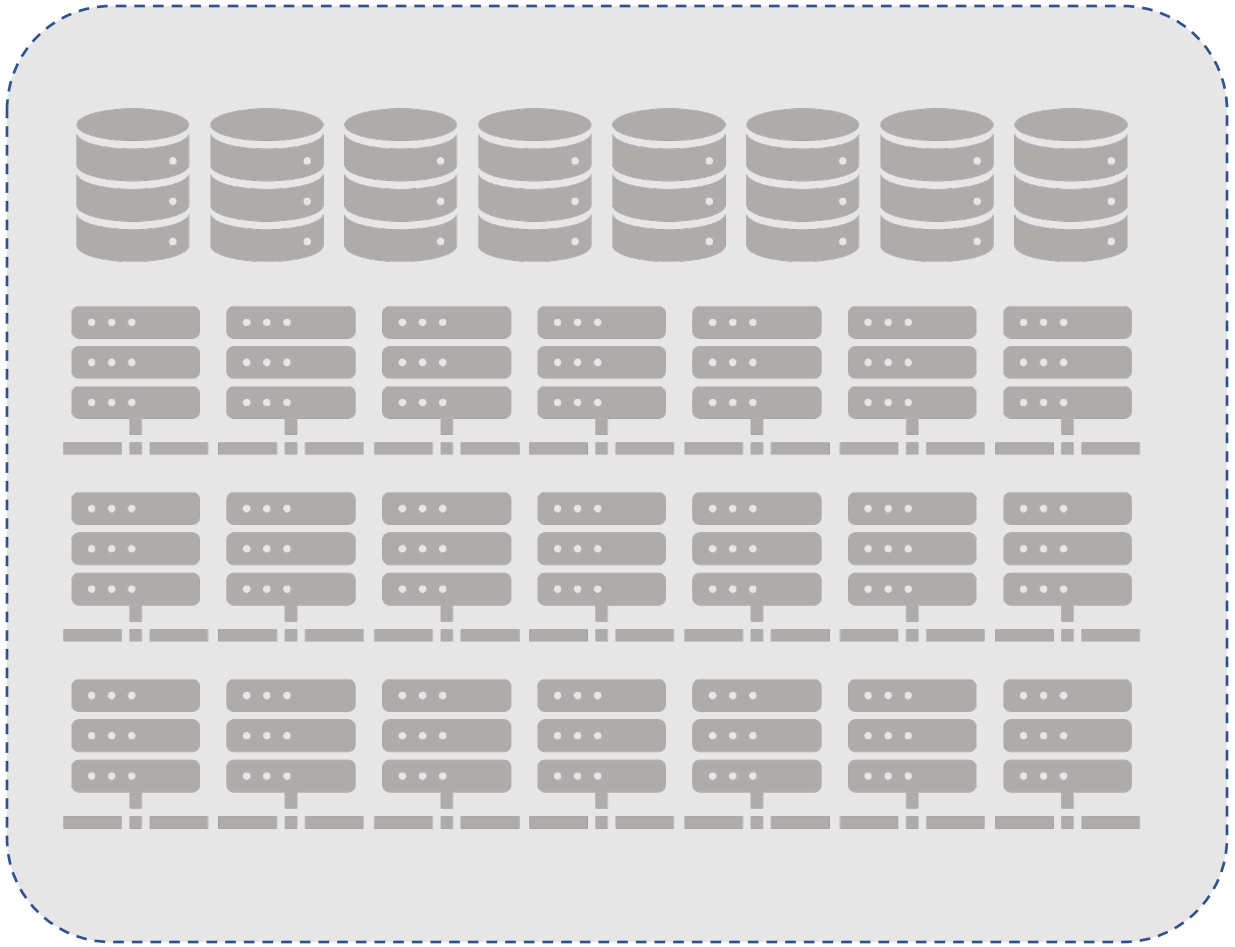
- **Job**
  - Sequence of ***Stages***
  - Triggered by an ***action***
  - collect(), read(), write() ...
- **Stage**
  - Sequence of ***Tasks***
  - Sequences run in parallel without a ***shuffle***
  - Output of an execution plan
- **Task**
  - Single operation applied to a partition
  - Triggered by a ***transformation***
  - map(), filter(), ...



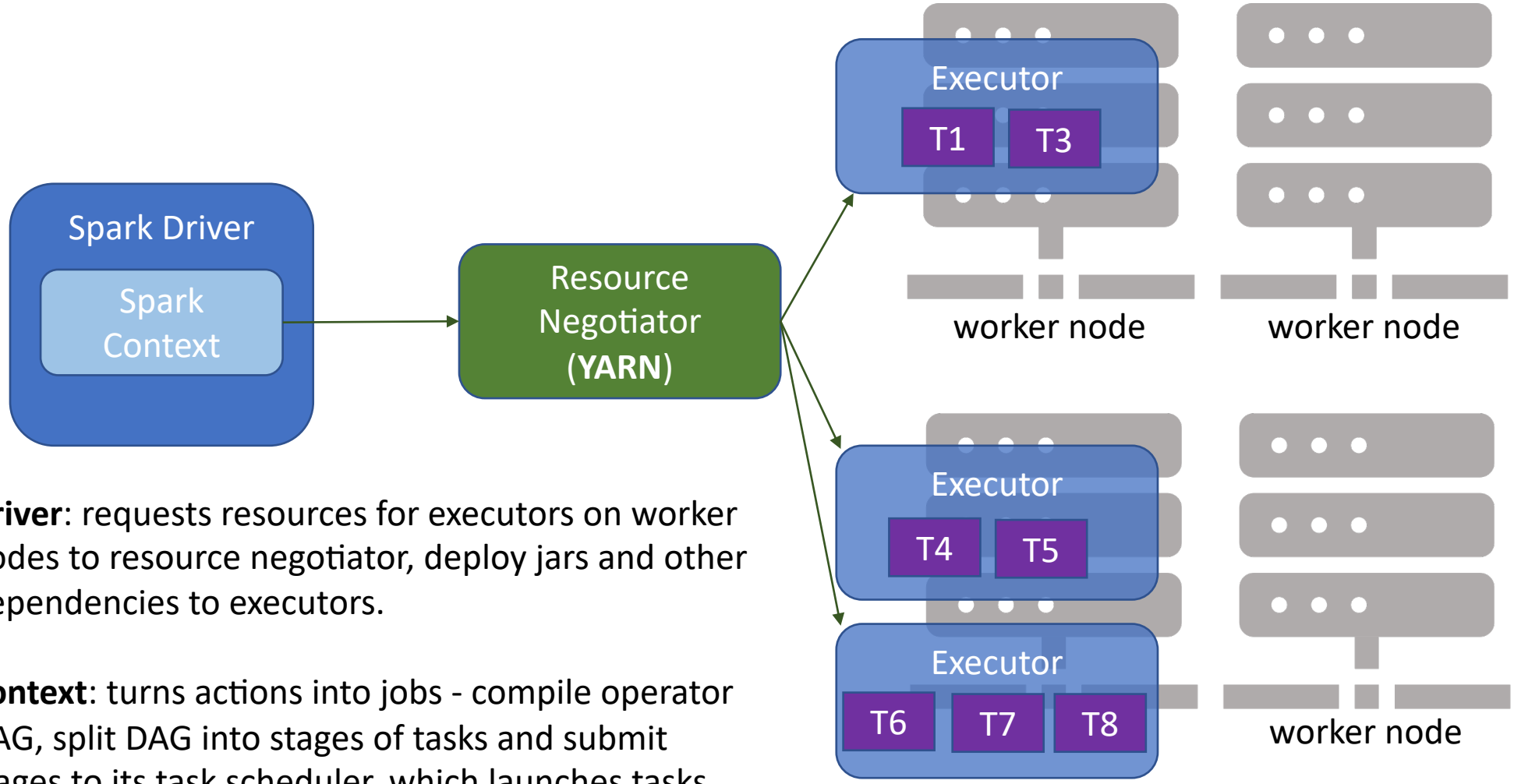
Spark and Big Data

Big Data and/or Big Jobs on big cluster





Spark Architecture - Revisited

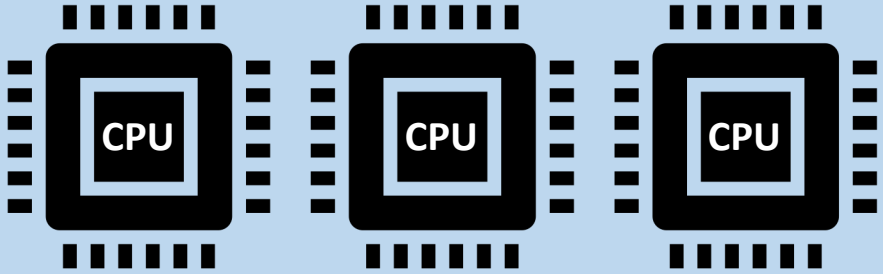


Driver: requests resources for executors on worker nodes to resource negotiator, deploy jars and other dependencies to executors.

Context: turns actions into jobs - compile operator DAG, split DAG into stages of tasks and submit stages to its task scheduler, which launches tasks.

Spark Executor – for heavy lifting

Executor



Number of cores

Memory

Execution 50%

In-memory storage 50%

"Local storage"

Handling Big Data

Time Efficiency

- Minimize data transfer
- locality principle:
  - Run with ‘local’ data
  - Avoid shuffle operations
- Minimize process time
  - Spark/PySpark optimizations
- **Tune partitions**

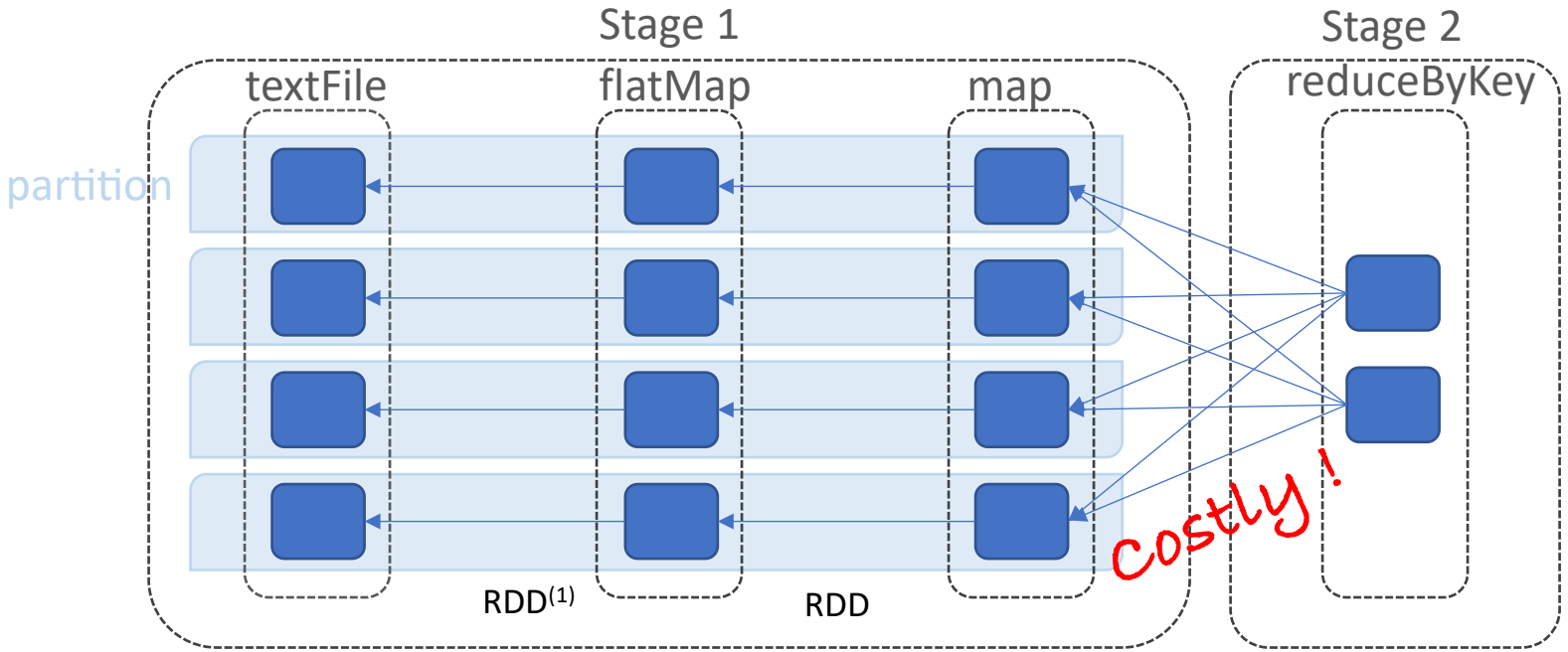
Space Efficiency

- Be aware of memory usage
- Minimize amount of data processed
- **Tune partitions**

Managing data shuffle and memory – best practices

Shuffle – what is it and how to avoid it

- Shuffle: all-to-all Costly !
- When: transfers data among executors
- E.g. `join, <reduce/sort/...> ByKey`



(1)RDDs point to parent

Why is Shuffling costly

- All-to-all data transfers - N^2
- Every transfer involves
  - Serialization data objects → bytes → data objects
  - Network I/O
  - Disk I/O
  - Operations run on a JVM
  - (often) Heap memory → spill to disk → more disk I/O

Shuffle Optimizations

- User optimization - avoid shuffle whenever possible
  - Thin the data early
    - Order the data or filter it in the partitions
  - Reorder transformations
    - When order of operations allow it, use operations that reduce the data the most
  - Partial aggregation

- Spark optimization
  - Write intermediate files to disk, so that the data does not get copied to all of the nodes

Memory

- Spark is written in Scala
  - JVM
  - (infamous) Garbage Collector

- When to suspect out of memory issues
  - Poor performance
  - Slow / dead executors (in unpredictable ways)

Memory - diagnostic (expert level)

- How to diagnose
  - set in spark.executor.extraJavaOptions
    - XX:+PrintGCDetails
    - XX:+HeapDumpOnOutOf
  - In linux, use `dmesg` for oom-killer logs
  - Spilling to disk

- If using YARN: take into account the memory constraints - executor could just be killed because they exceed their allocated capacity

Memory optimization

- Avoid spilling to disk and OOM deaths
- Aim for key-value that fit in memory
  - Number of keys – if too large, it may kill the driver (in reduce stage)
  - Number of values per keys – if too large it may kill executors
- Adjust partition size
  - Increase number of partitions
  - Decrease size of partitions
- Minimize shuffle, which is also memory intensive

Partitioning

How is partitioning done? Number of partitions

- Automatically
  - HDFS - partition by HDFS block
  - Spark default partitions

```
sc.defaultParallelism()
rdd.getNumPartitions()
```

- Programmatically
  - Create RDD with custom number of partitions

```
sc.parallelize(data, num_partitions)
```

  - Change the number of partitions in next RDD

Preferred

```
rdd.repartition(num_partitions)
rdd.repartitionAndSortWithinPartitions(num_partitions,...)
rdd.coalesce(num_partitions,shuffle=false)
```

Configure partitions

Input - control size

- spark.default.parallelism()
- spark.sql.files.maxPartitionByBytes(b)

Shuffle - control count

- spark.sql.shuffle.partitions(n)

Output - control size

- coalesce(n) to shrink
- repartition(n) to increase and/or balance
- df.write.option(“maxRecordsPerFile”, N)

Optimum number of partitions

Too few partitions

- Less concurrency
- more susceptible to data skew
- increased memory pressure

Too many partitions

- Less concurrency
- more susceptible to data skew
- increased memory pressure

Right balance

- 2 – 4x number of cores
- Aim for 100ms of executions at least

- Understand data access pattern: e.g. write once, read many
- Coalesce partitions to size-down file set
- Repartition = another stage = expensive
  - `df.repartition(n)` HashPartitioner
  - `df.repartition(n,[colA,..])` RangePartitioner
- df.localCheckpoint + repartition or coalesce
  - stage barrier

Spark Partitioners

- HashPartitioner
  - Hash code of an object
- RangePartitioner
  - Sortable records
- CustomPartitioner
  - Extend Partitioner
  - Overwrite methods
  - And define the comparison method

```
def numPartitions: Int
def getPartition(key: Any): Int
```

```
def equals(other: Any): Boolean
```

Spark Partitioners - RDD

- HashPartitioner: pyspark.RDD.repartition

```
newRdd = oldRdd.repartition(numPartitions)
```

- Custom partitioner: pyspark.RDD.partitionBy

```
def partitioner(key):
        if key == 'foo':
              return 1
        else:
             return random.randint(2,numPartitions)

newRdd = oldRdd.repartitionBy(numPartitions,partitioner)
```

PySpark will use: partitioner(key) modulo numPartitions

Spark Partitioners - DataFrame

- HashPartitioner: pyspark.sql.DataFrame.repartition
  - Hash code of an object

```
newDf = oldDf.repartition(numPartitions,"colA","colB")
```

- RangePartitioner: pyspark.sql.DataFrame.repartitionByRange

```
newDf = oldDf.repartitionByRange(numPartitions,"colA","colB")
```

Optimization check list

- Ensure you have enough partitions, but that partitions are not underused
- Optimize placement of functions
  - Use the right functions at the right place
- Choose partitioning functions wisely
  - Aim for well balanced partitions, avoid partition skew
  - Minimize shuffling and amount of data shuffled
- Minimize memory consumption, i.e. memory hungry functions (sorting, groupBy)
  - Can cause OOM

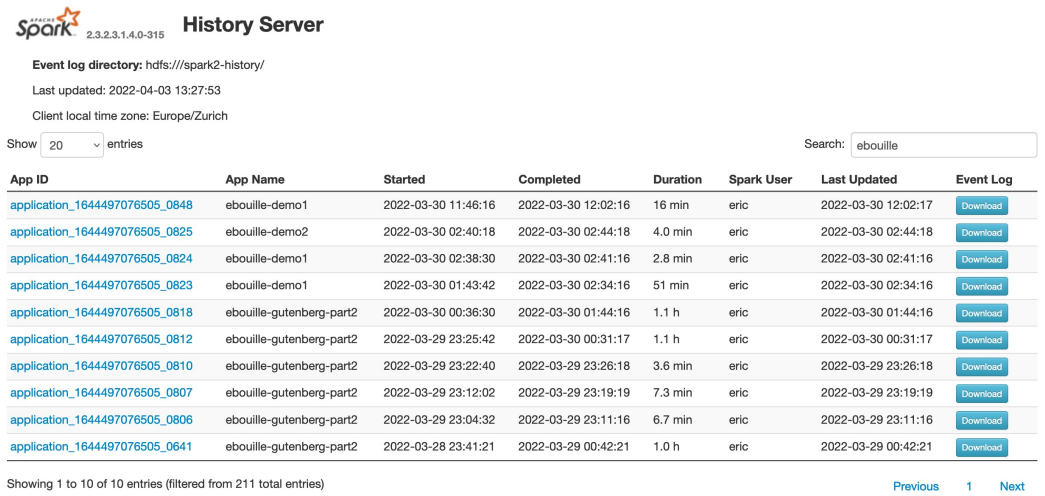
Spark & YARN Troubleshooting

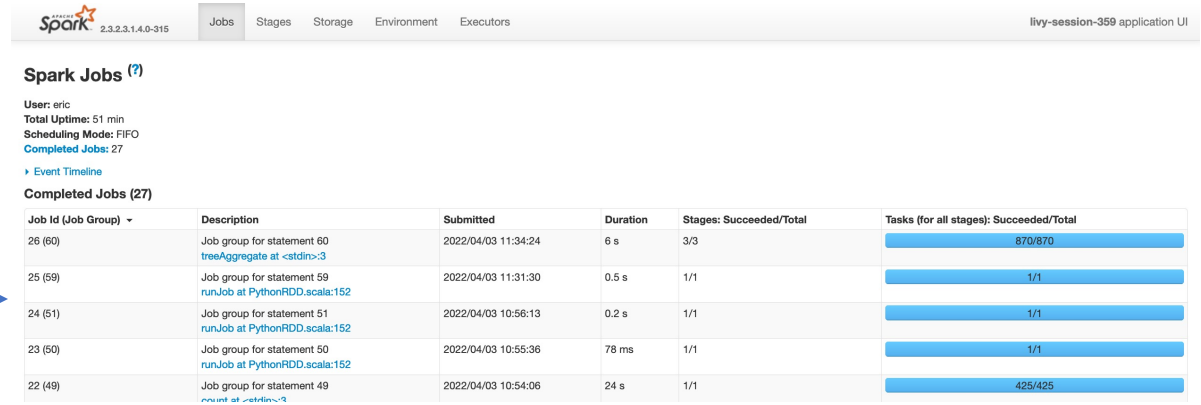
Spark UI

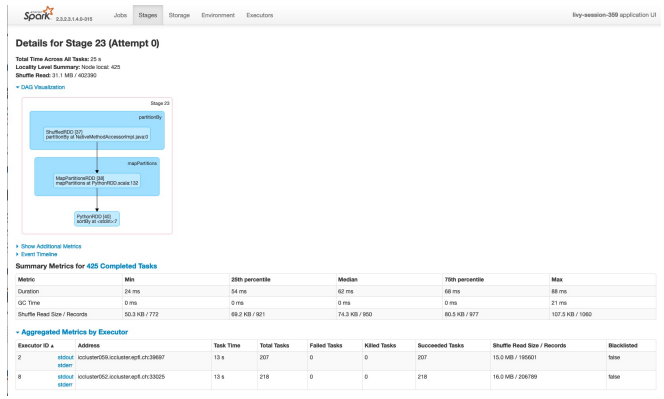
- Use Spark UI to tune tasks/partitions
- See stages, see tasks (#of tasks, shuffle read sizes)
- See failed tasks, tasks that are taking longer
- Collect and read standard output and error logs
- Useful for shared clusters if you are not admin

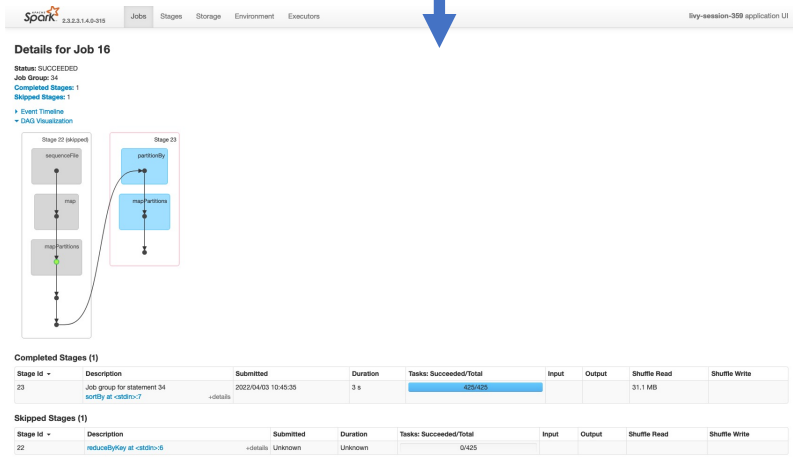
Spark UI – COM490 Spring 2022

http://iccluster029.iccluster.epfl.ch:18081/







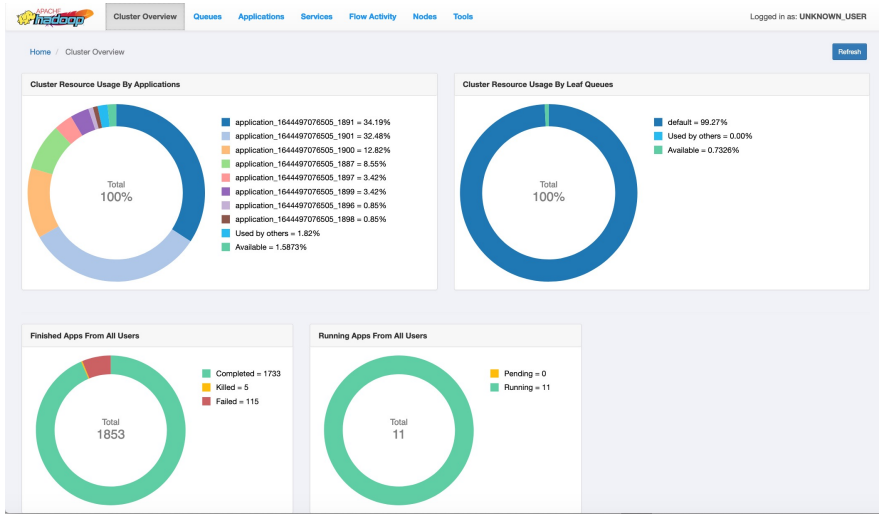


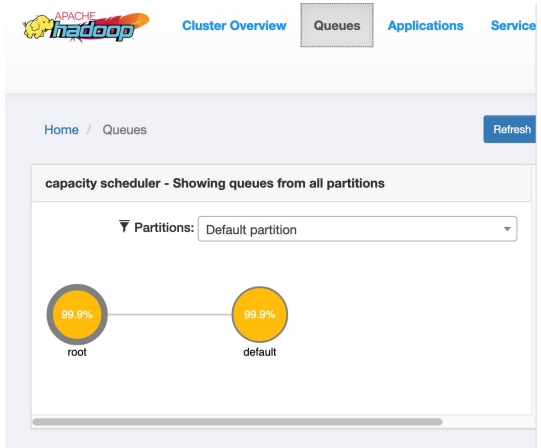
Applications
- ↘ **Jobs**
  - ↘ **Stages**
    - ↘ **Tasks**

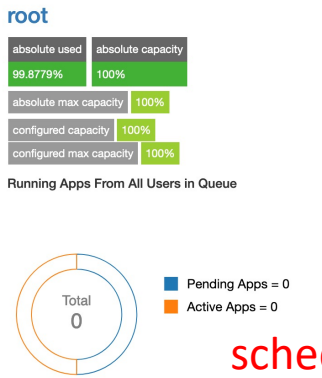
YARN UI – COM490 Spring 2022

http://iccluster029.iccluster.epfl.ch:8088/ui2/#/cluster-overview

resource usage

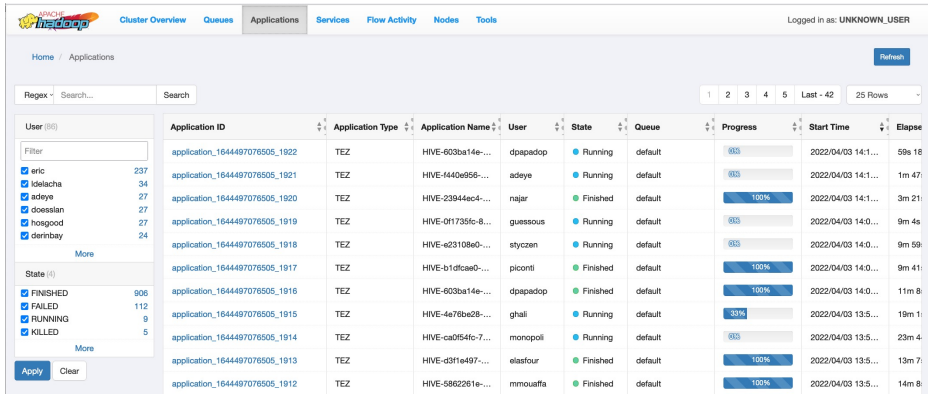




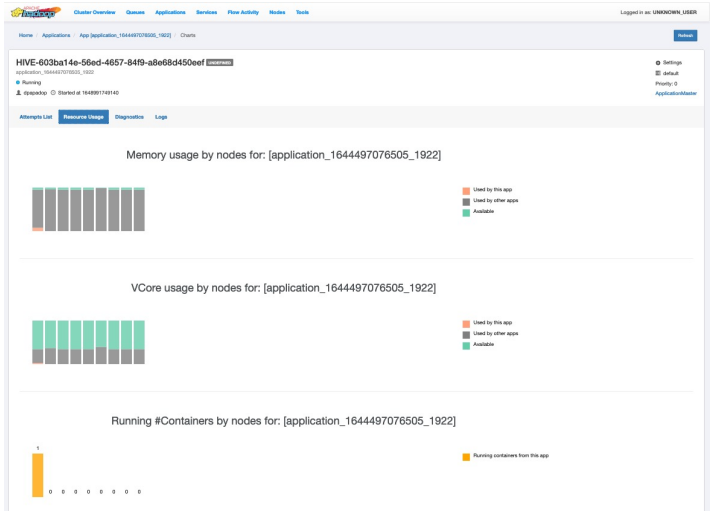


scheduler queues

applications



application usage, logs



Troubleshooting – useful YARN commands

- yarn app -list
- yarn logs -help
  - yarn logs -applicationId {applicationID}
- yarn top
- yarn queue -status default
- yarn node -showDetails -list -all

Don’t try other commands, or ask. They can be potentially dangerous

Note: you should enter the commands manually, because some of the text are sometime automatically converted into forms that cannot be interpreted by the command line

Know the infrastructure (expert)

- Memory and cores per VM / Worker node
- Speed per core
- Network speed
- Disk capacity and disk type (SSDs, remote)

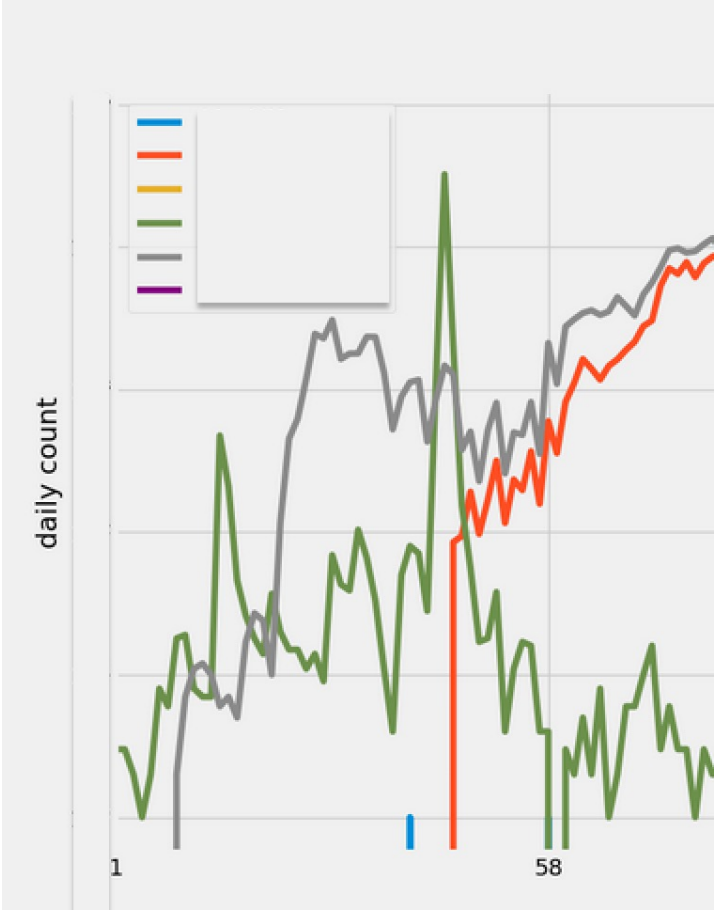
- Goal: improve utilization, aim for 70%

Start your engines

https://dslab2022-renku.epfl.ch/projects/{yourname}/lab-course

Homework 3

- A year of Twitter events from 2020
- A tumultuous year that saw the onset of a deadly pandemic, widespread protests, and a heated election
- Can you reconstruct the events from social micro-bloggings?



GRADED ASSIGNMENT 3

- Due **26.04.2021 23:59:59**
- Checklist
  - Create your group in gitlab (contact us if you do not have a group)
  - Fork homework-3 under your group name
    - https://dslab2022-renku.epfl.ch/projects/com490/homework-3
    - Answer questions in homework's notebook
  - Work in team (take advantage of git, gitlab and renku)
  - Only commit the code and comments
    - Do not commit results of notebook execution (in history or in final version)
      - Do a cell output clear before committing
  - Always check your result
    - Restart kernel and run all cells (then clear the cells)
  - **Submit before deadline: git add + git commit + git push**