Optimisation - Spark - ESGI 2020

Agenda .

1. Spark UI & History Server
2. Warmup
3. Exercise 1 - RDD, cache monitoring and shuffle
4. Exercise 2 - Paris will always be the first
5. Exercise 3 - DataFrame bad practices
6. Exercise 4 - Partitioning
7. Exercise 5 - Spark problem analysis
8. Exercise 6 - Joins
9. Exercise 7 - Custom listener (Bonus)

Version of the frameworks & tools .

Apache Spark

2.4

Scala

2.11

Apache Maven

3.x

Spark UI & History Server .

Spark UI .

- Motinor **Spark processing**, **Cache utilization**, **RAM** & **GC Utilization**
- **URL:** http://localhost:4040
- **Job**
  - Sequence of transformations, finished by an action
  - Composed by **Stages**
- **Stage**
  - Sequence of transformation without repartition (no shuffle)
  - Composed by parallel **Tasks**
- **Task**
  - Smallest processing unit
  - For one partition / one Thread

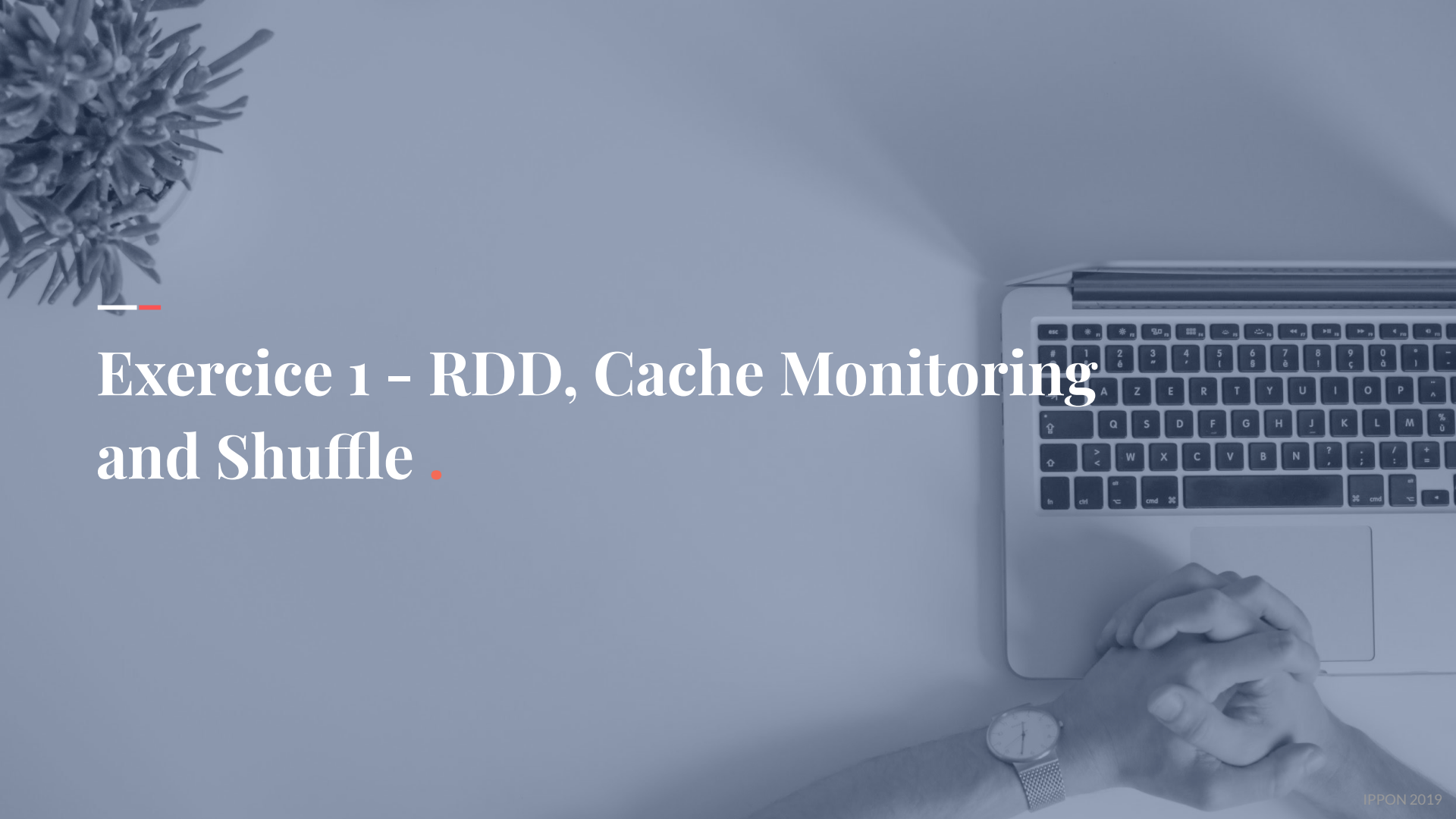
Spark UI - Tabs .

- **Jobs**
  - Visualize list of jobs
- **Stages**
  - Visualize list of stages
- **Storage**
  - Statistics on caches
- **Environment**
  - Environment variables & Spark's Configuration
- **Executors**
  - Monitors executors
- **SQL**
  - Visualize DataFrames processing

History Server .

- ★ Consult Spark's logs once the job ended
- ★ How to activate
  - Step 1 : Edit spark-default.conf
    - cp conf/spark-defaults.conf.template conf/spark-defaults.conf
    - vi conf/spark-defaults.conf
    - spark.history.fs.logDirectory=<root-directory>/src/main/datasets/spark-events
  - Step 2 : Job options (already done)
    - spark.eventLog.enable = true
    - spark.eventLog.dir = <root-directory>/src/main/datasets/spark-events
- ★ URL: http://localhost:18080

Exercice 1 - RDD, Cache Monitoring and Shuffle .



Objectives .

1. How can we monitor the cache usage in Spark?
2. What is Shuffle and how handle it?
3. How to optimize and speed up our job?
4. How read the Spark UI?

Preparation .

1. Import the codingdojo.zip in a working spark project
2. Change package name for files
3. Change path in Constants.scala
4. Make sure you have download the dataset. https://www.insee.fr/fr/statistiques/fichier/3141877/RP2014_logemt_txt.zip

Use case

- You need to produce a csv file with three fields:
  - Commune: the name of the city
  - nb_voiture: Number of cars registered in this city
  - average_moving: Average year of moving-in
- You already have the source dataset (FD_LOGEMT_2014)
- A coworker had already produced some code running in production, but doesn't really work quite well…
- You need to optimize this code!

Cache Monitoring .

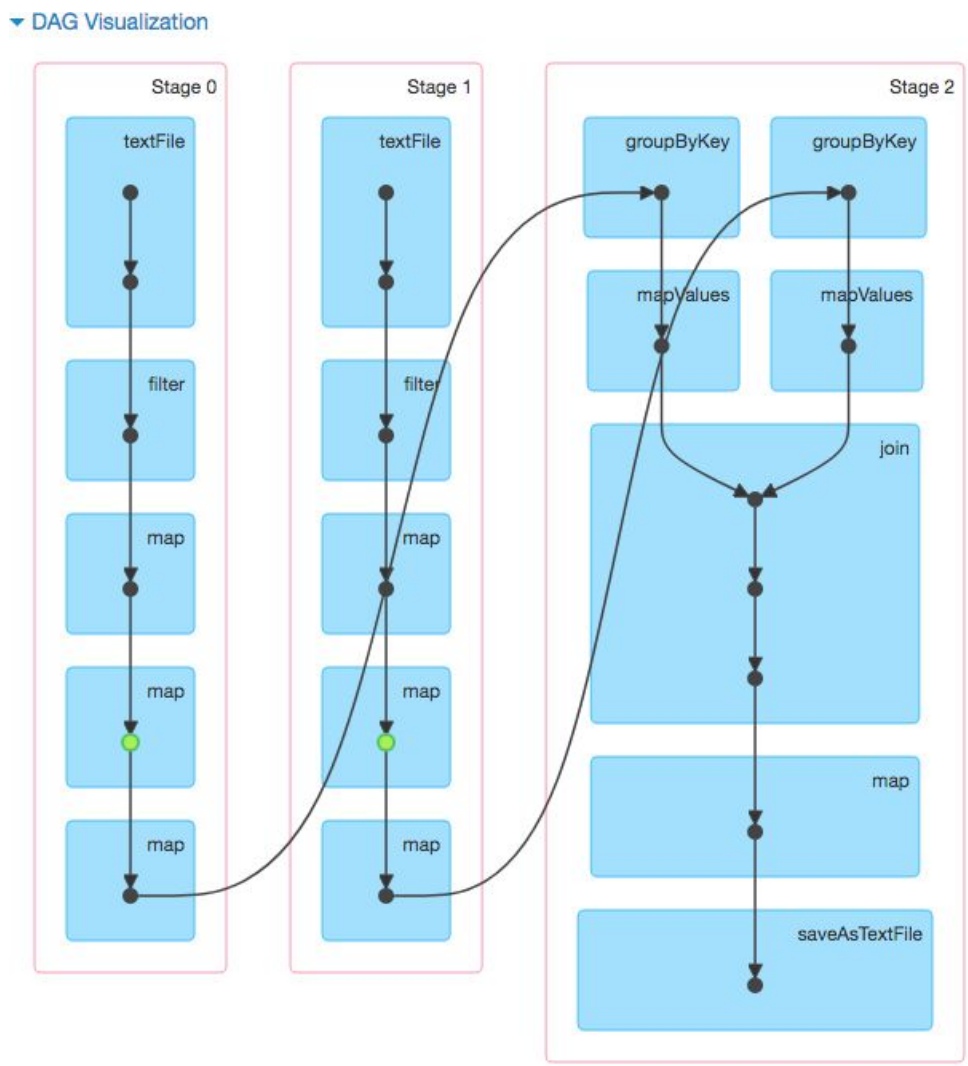
- The Exercice1 program compute for each city (COMMUNE)
  - The car number
  - The average year of moving
- Read then execute the job. Look at the Spark UI

- Is the cache usefull?
  - Read the code then look at the DAG for job 0 on Spark UI (green point)

- Is the cache efficient?
  - Read the statistics of cache and executors

Is the cache useful ?

DAG .


DAG Visualization
Stage 0
textFile
filter
map
map
map
Stage 1
textFile
filter
map
map
map
Stage 2
groupByKey
groupByKey
mapValues
mapValues
join
map
saveAsTextFile

Is the cache efficient ?

Executors .

- ★ The job reads 14 GB of data
- ★ We have an alert on the Garbage Collector

Reminder, the Garbage Collector is the memory recycling component. It clean the unused RAM.

Complete Tasks	Total Tasks	Task Time (GC Time)	Input	Shuffle Read
2417	2417	50 min (28 min)	14 GB	261.3 MB
0	0	0 ms (0 ms)	0.0 B	0.0 B
2417	2417	50 min (28 min)	14 GB	261.3 MB

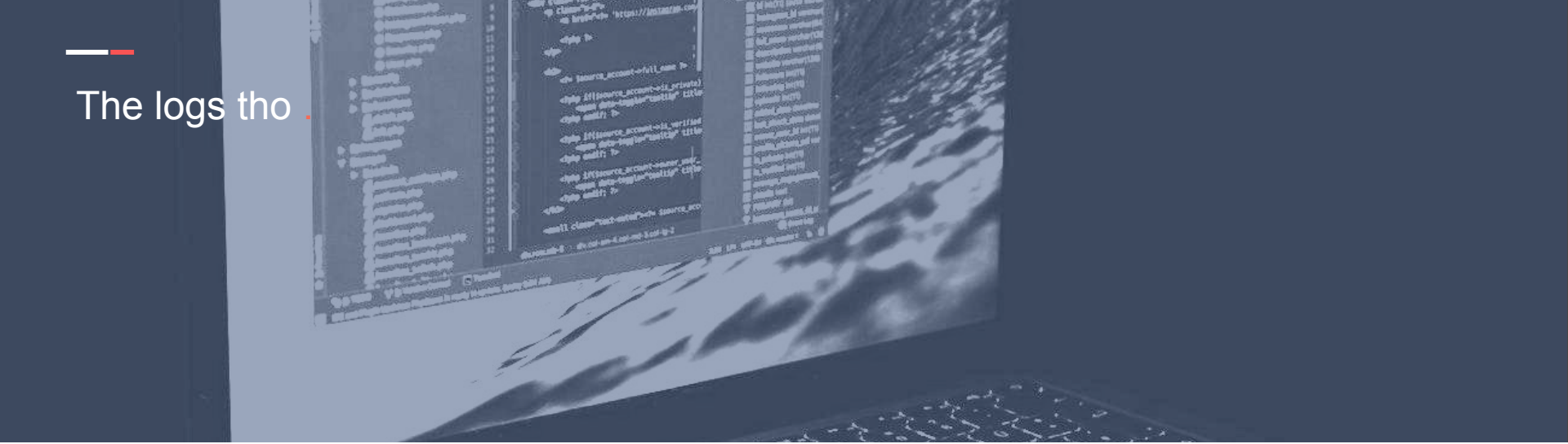
Storage .

The tab display cache usage

Storage

RDDs

ID	RDD Name	Storage Level	Cached Partitions	Fraction Cached	Size in Memory	Size on Disk
4	MapPartitionsRDD	Memory Deserialized 1x Replicated	2	1%	1234.1 MB	0.0 B



The logs tho .

WARN [Executor task launch worker for task 18] org.apache.spark.storage.memory.MemoryStore - Not enough space to cache rdd_4_18 in memory! (computed 106.4 MB so far)

WARN [Executor task launch worker for task 18] org.apache.spark.storage.BlockManager - Block rdd_4_18 could not be removed as it was not found on disk or in memory

WARN [Executor task launch worker for task 18] org.apache.spark.storage.BlockManager - Putting block rdd_4_18 failed

Analysys .

Elements :

- ★ Cache seems mandatory but only few data is cached
- ★ Garbage Collector is used very heavily

Hypotesis:

- ★ Spark cannot cache all of the data in memory
- ★ It spends too much time loading/unloading data from/to cache

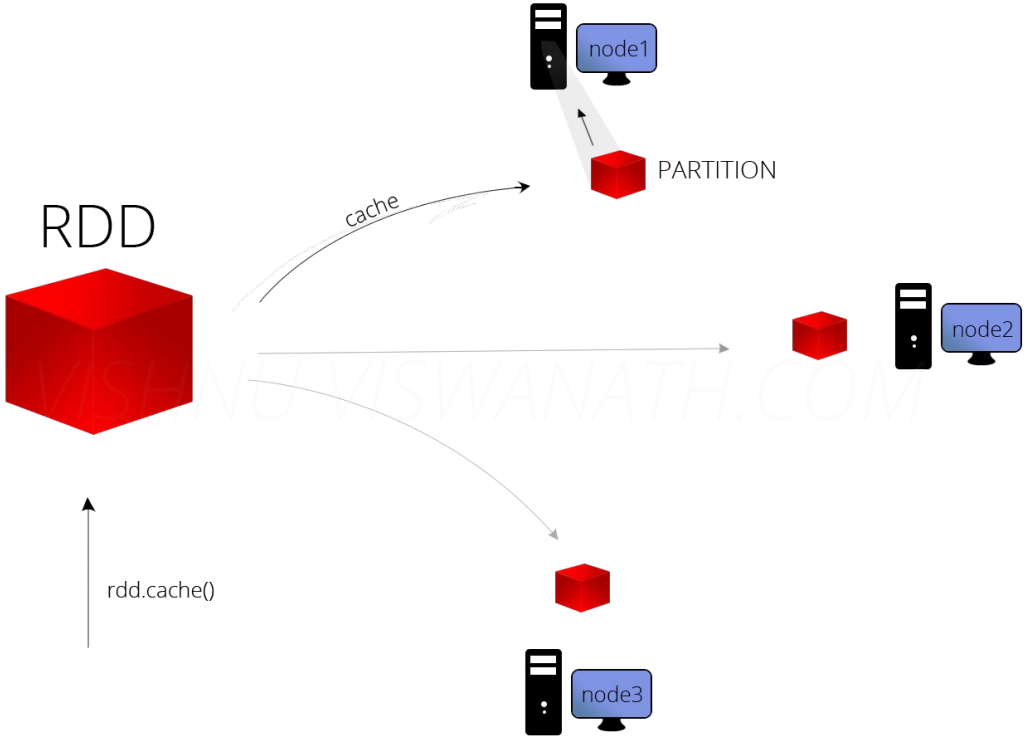
Adaptation 1/2 - Caching .

- ★ Read cache and persist documentation
- ★ Try different StorageLevel
  - MEMORY_ONLY
  - MEMORY_ONLY_SER
  - MEMORY_AND_DISK
  - MEMORY_AND_DISK_SER

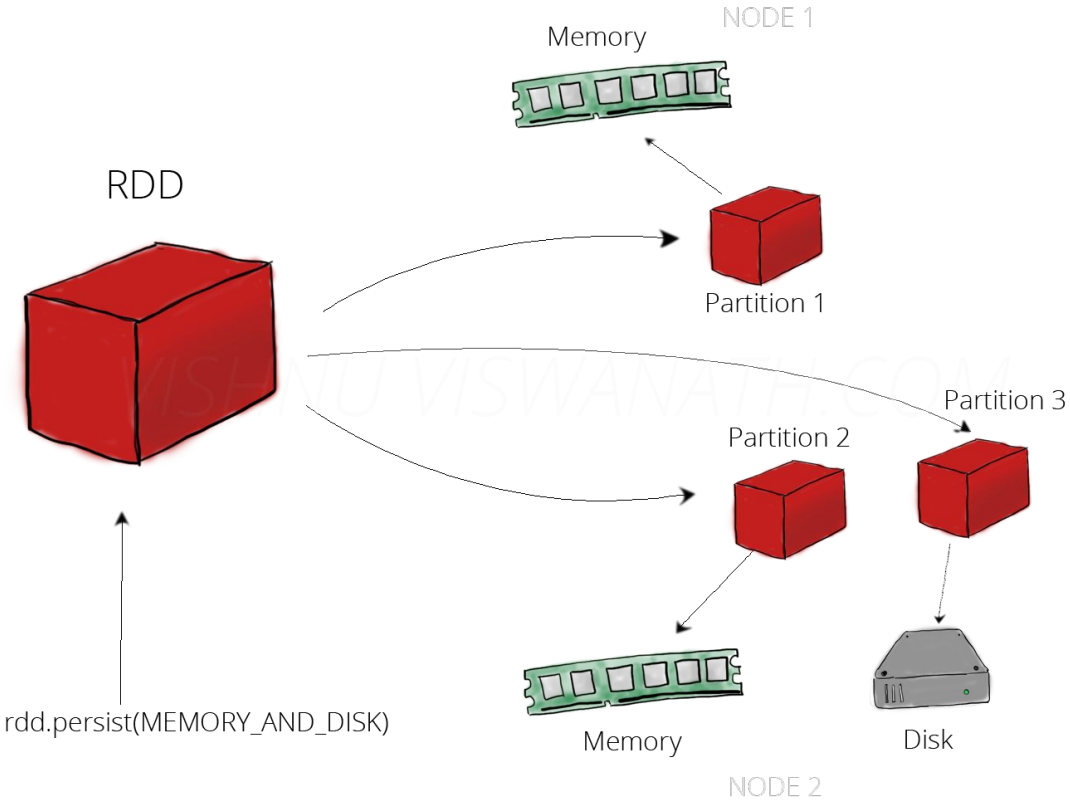
Reminder:

- ★ Deserialized Storage→ Java instances are directly written
- ★ Serialized Storage → Objects are stored as Byte Array

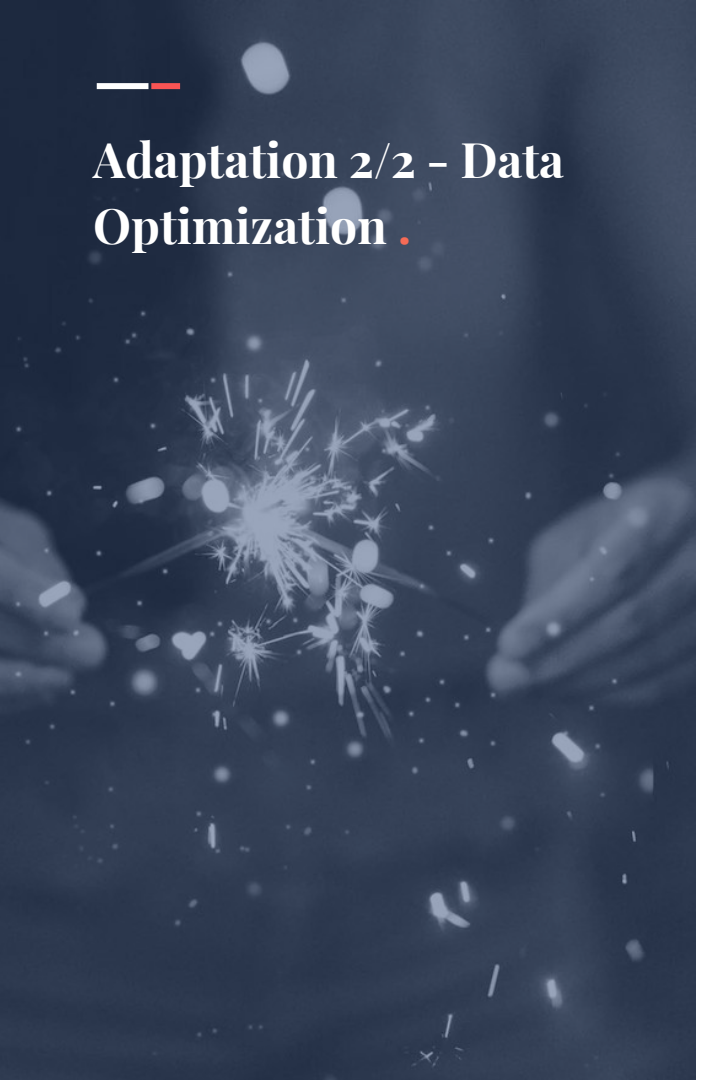
Adaptation 1/2 - Caching .



Adaptation 2/2 - Caching .



Are all fields mandatory ?



Adaptation 2/2 - Data Optimization .

Adaptation:

- ★ Write the “Enregistrement_simple” class to only use mandatory fields for our job
- ★ Map “Enregistrement” to “Enregistrement_simple”

Data are cached in string

- ★ Convert fields VOIT and AEMM in integer in the class “Enregistement_Simple”

Spark can use Kryo library to speed up serialization

- ★ Declare “Enregistrement_simple” to Kryo

  Tips: sparkContext.registerKryoClasses

Adaptation - Results .

Execution time decreased

Adaptation:

- ★ Changing cache strategy
- ★ Storing only needed data
- ★ Use of Kryo serializer to speed up the job

Going further

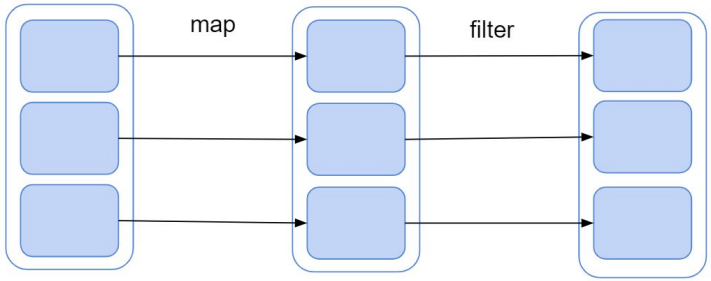
Spark Operations .

Some methods aren’t quite optimized for our use case...

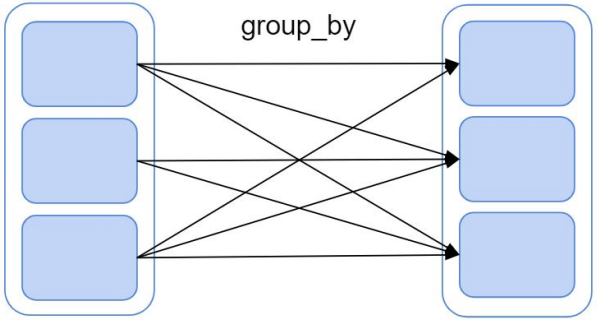
Spark Operations .

We can classify transformations in 2 categories:

Narrow Transformation : Transform where all needed element for the compute are on only one parent partition (Map, Filter…)



Wide Transformation : Transform where all needed element for the compute are on multiple parent partition(reduceByKey, join, …)



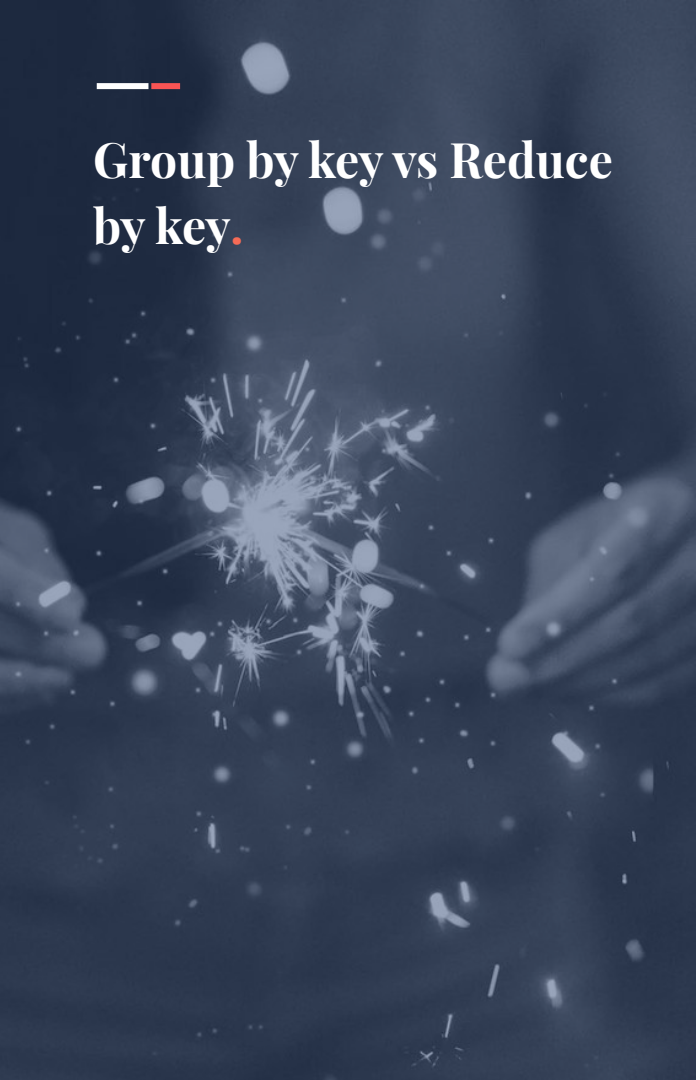
Shuffle .

Shuffle:

- ★ Mechanism of data redistribution between cluster partitions. Executed before a wide transformation.

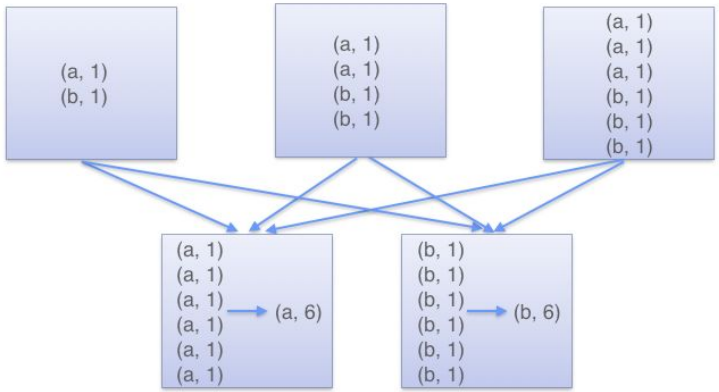
Impacts:

- ★ Expensive cost
- ★ Disk I/O
- ★ Object serialization
- ★ Network I/O

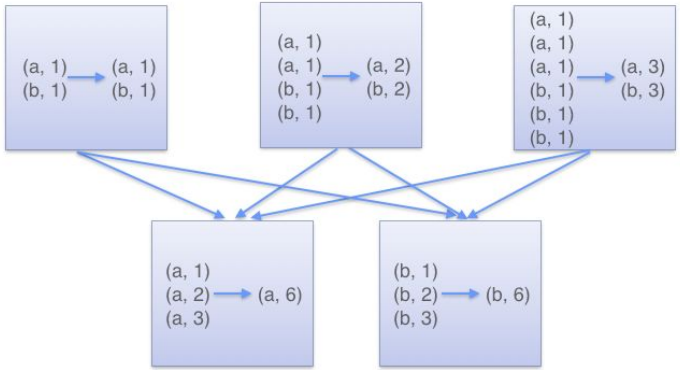


Group by key vs Reduce by key.

GroupByKey

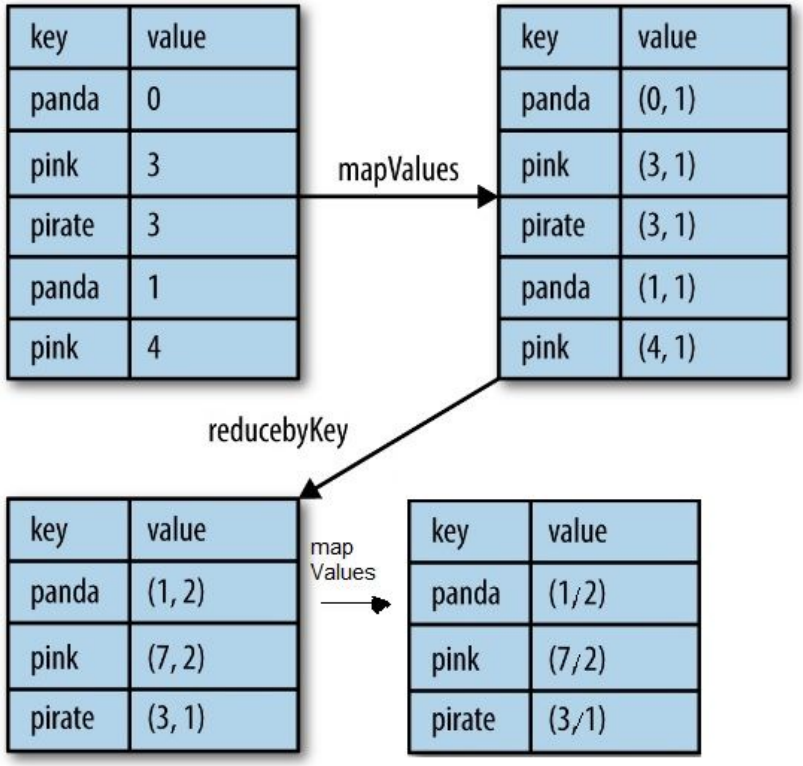


ReduceByKey



Group by key vs Reduce by key.

★ Replace groupByKey by ReduceByKey in the exercise.



Exercice 1 - Conclusion.

What we have done:

- ★ Look at information in Spark UI
- ★ Cache :
  - Analyze cache usage
  - What are the cache strategies
  - Recommendations :
    - MEMORY_ONLY
    - MEMORY_ONLY_SER
    - MEMORY_AND_DISK_SER
- ★ Data optimization
- ★ Shuffle :
  - What is Shuffle
  - Avoid groupByKey when you can

Exercise 2

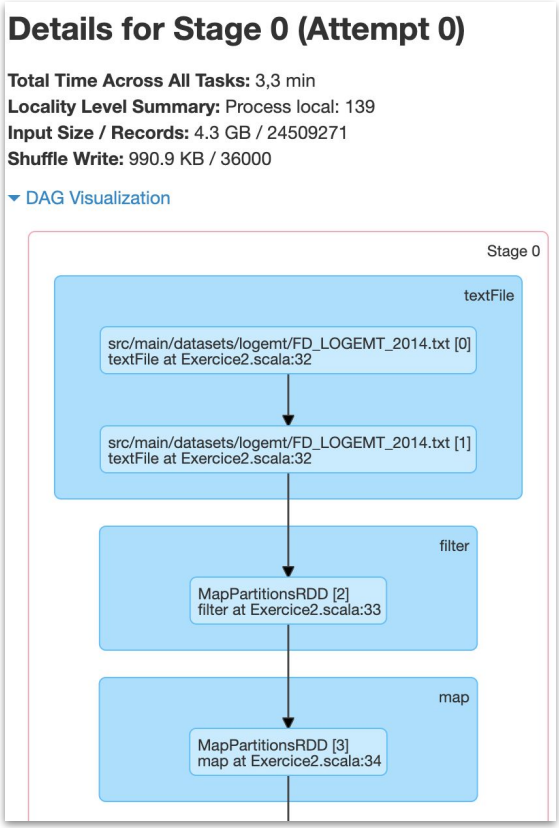
Exercise 2

- Compare RDD and Dataframe implementation of the same job

- Open Exercise 2:
  - The job compute the number of cars by city in France
  - Read then launch Exercice2 job
  - Convert this job using DataFrames
  - Compare execution time using history Server

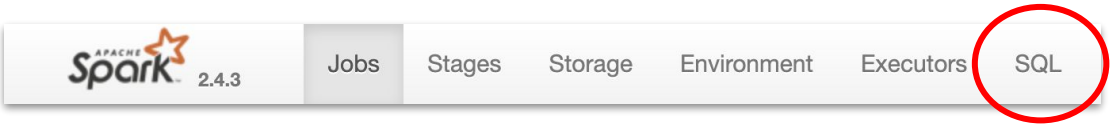
Exercise 2

★ With RDDs and text files, Spark seems to read all of our dataset, even tho we only need a few parts of it

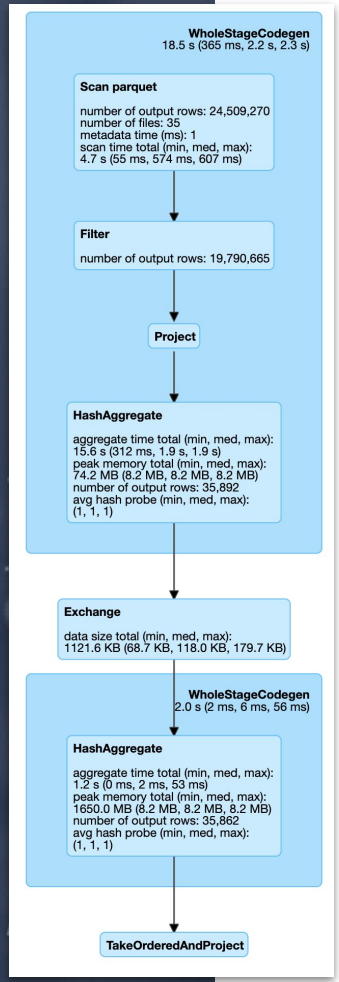


Exercise 2

- Since we used DataFrames, we have a new tab in Spark UI
- It contains the Logical Query plan

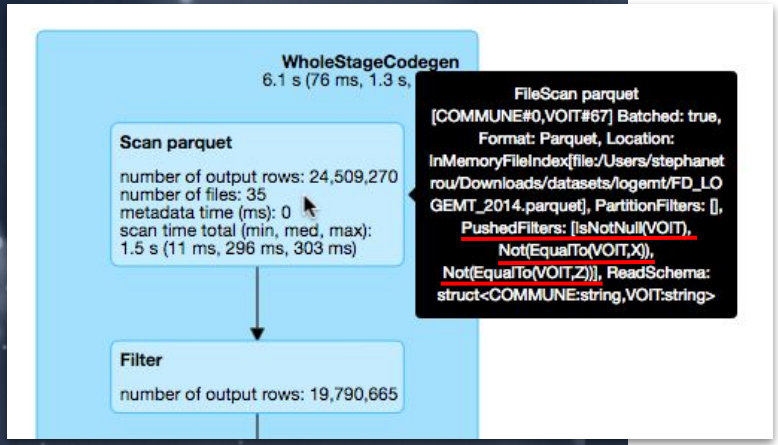


Exercise 2



- The Logical Query Plan Contains way more informations than the classic DAG

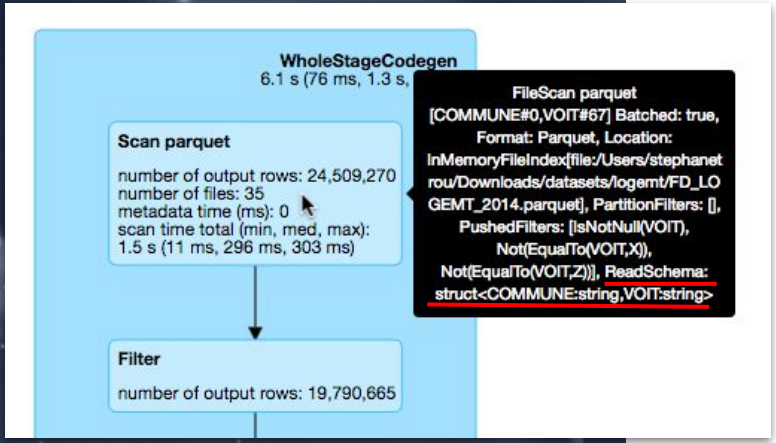
Exercise 2 - PushDown Filter.



Spark's DataFrame engine detected our filtering conditions on VOIT column

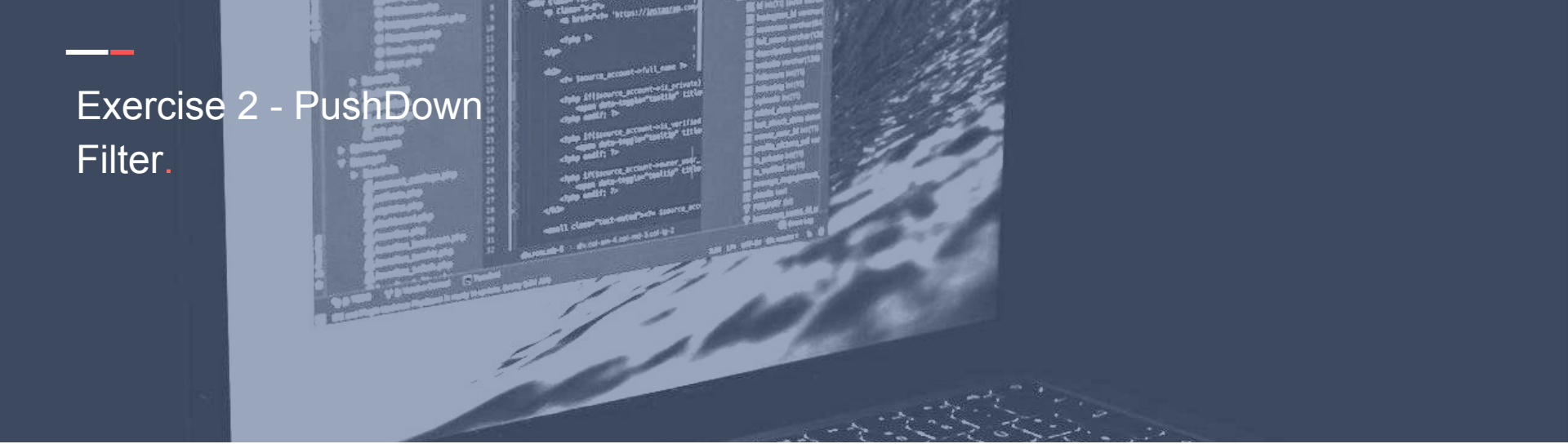
- → It has loaded in memory the result of this filtering condition
- This is: PushDown Filter

Exercise 2 - PushDown Filter.



Spark's DataFrame engine detected that we only needed to load COMMUNE and VOIT column

- → It has loaded in memory only these two columns
- This is: PushDown Projection



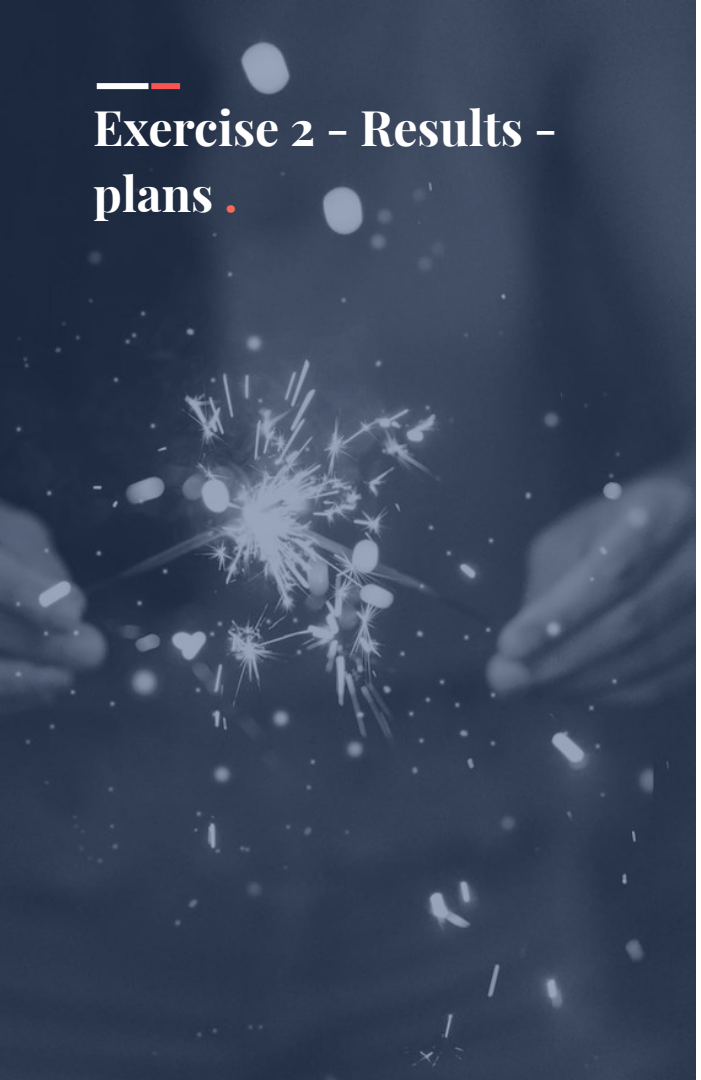
Exercise 2 - PushDown Filter.

This information is also accessible in the Physical Plan

```
== Physical Plan ==
TakeOrderedAndProject(limit=21, orderBy=[NB_VOITURES#278L DESC NULLS LAST], output=[COMMUNE#0,NB_VOITURES#284])
+- *(2) HashAggregate(keys=[COMMUNE#0], functions=[sum(cast(VOIT#138 as bigint))], output=[COMMUNE#0, NB_VOITURES#278L])
   +- Exchange hashpartitioning(COMMUNE#0, 200)
      +- *(1) HashAggregate(keys=[COMMUNE#0], functions=[partial_sum(cast(VOIT#138 as bigint))], output=[COMMUNE#0, sum#288L])
         +- *(1) Project [COMMUNE#0, cast(VOIT#67 as int) AS VOIT#138]
            +- *(1) Filter ((isnotnull(VOIT#67) && NOT (VOIT#67 = X)) && NOT (VOIT#67 = Z))
               +- *(1) FileScan parquet [COMMUNE#0,VOIT#67] Batched: true, Format: Parquet, Location: InMemoryFileIndex[file:/Users/williamconti/Documents/spark-training-enonces/src/main/datasets/log..., PartitionFilters: [], PushedFilters: [IsNotNull(VOIT),
Not(EqualTo(VOIT,X)), Not(EqualTo(VOIT,Z))], ReadSchema: struct<COMMUNE:string,VOIT:string>
```

The logical query plan can be accessed in the code

```
dataFrame.explain(extended = true)
```



Exercise 2 - Results - plans .

- **Parsed Logical Plan**: Raw tree describing all the operations
- **Analyzed Logical Plan**: Request tree where tables, columns, types are validated
- **Optimized Logical Plan**: Logical tree in which operations are optimized, combined and replaced
- **Physical Plan**: Tree describing where operations will be executed

Exercise 2 - Conclusion.

Using **DataFrames** and **parquet**, we have automatic optimisation, allowing us to obtain much **better performances** rather than using RDD and trying to optimize ourselves.

Exercise 3 - Dataframe bad practices

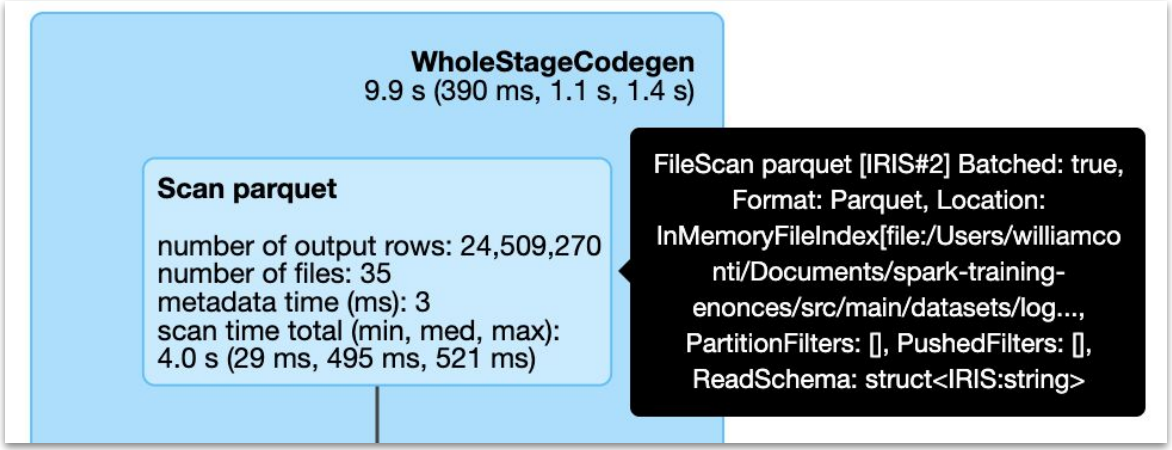
Exercise 3 - Objectives.

Avoid bad practices with DataFrames

- ★ Count the number of accomodations near the building Défense Ouest
- ★ Read and launch the Exercise 3
- ★ Examine the differences between the two jobs

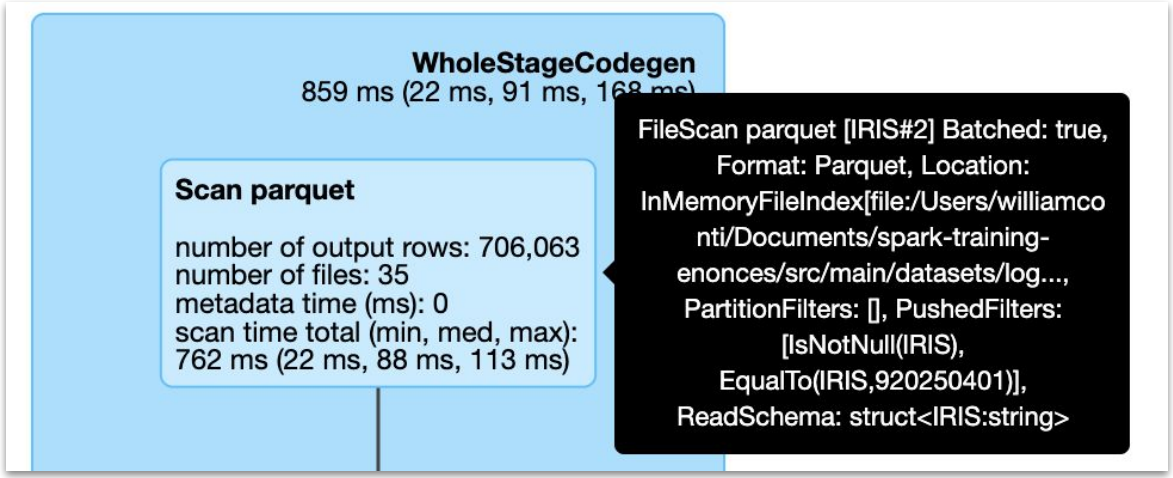
Exercise 3 - Conclusion.

First Job have PushDown projection but doesn't have PushDown filter



Exercise 3 - Conclusion.

Second Job have PushDown projection and filter



Exercise 3 - Conclusion .

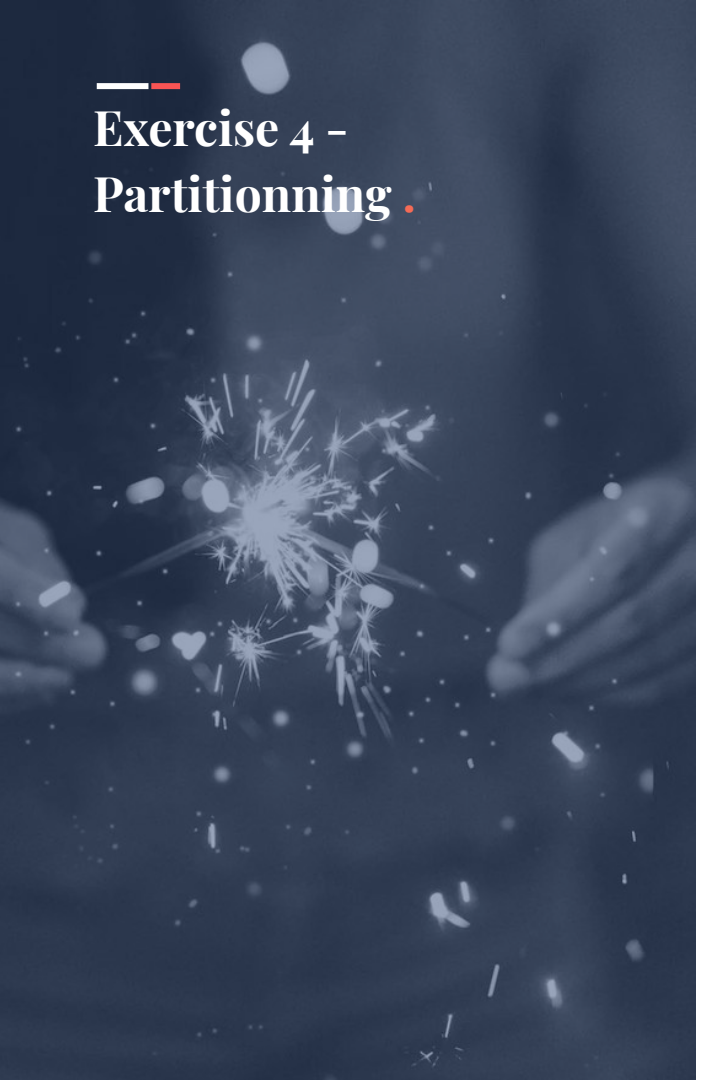
Job Id ▾	Description	Submitted	Duration
2	Count par Column depuis un fichier fr.ippon.training.spark.codingdojo.configuration.Constants$@57bdceaa.LOGEMENT_PARQUET count at Exercice3.scala:27	2019/06/03 15:11:18	0,2 s
1	Count par UDF depuis un fichier fr.ippon.training.spark.codingdojo.configuration.Constants$@57bdceaa.LOGEMENT_PARQUET count at Exercice3.scala:24	2019/06/03 15:11:16	2 s

This is why job 2 is faster than job 1

Exercise 3 - Conclusion

- Avoid pre-optimizations:
  - Unless you know what you are doing
- When you want to do something, use Spark's built-in methods when possible
  - groupByKey + mapValues is an example of bad knowledge of the API

Exercise 4 - Partitioning .

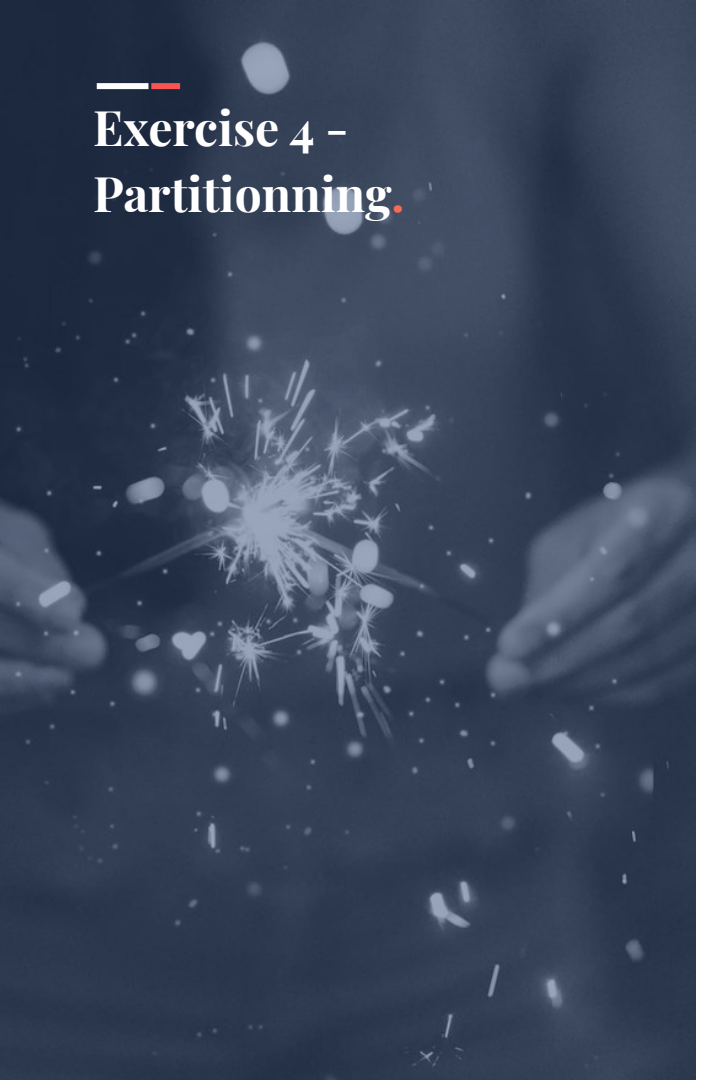


Exercise 4 - Partitionning .

The number of partitions influence the execution time of a job

- ★ Read and launch the Exercise 4
- ★ Play around with different partition number
  - Try with 1, 50 , 100, 1000, 10000

- ★ `spark.sql.shuffle.partitions`
  - Number of partitions before a shuffle
- ★ `spark.default.parallelism`
  - Number of partitions for distributed “reduce” operations and for default rdd parallelism



Exercise 4 - Partitionning.

- If a DataFrame has too many partitions, then task scheduling may take more time than the actual execution time.
- Having too less partitions is also not beneficial as some of the worker nodes could just be sitting idle resulting in less concurrency.

Exrcise 4 - Partitionning.

Tasks (9)

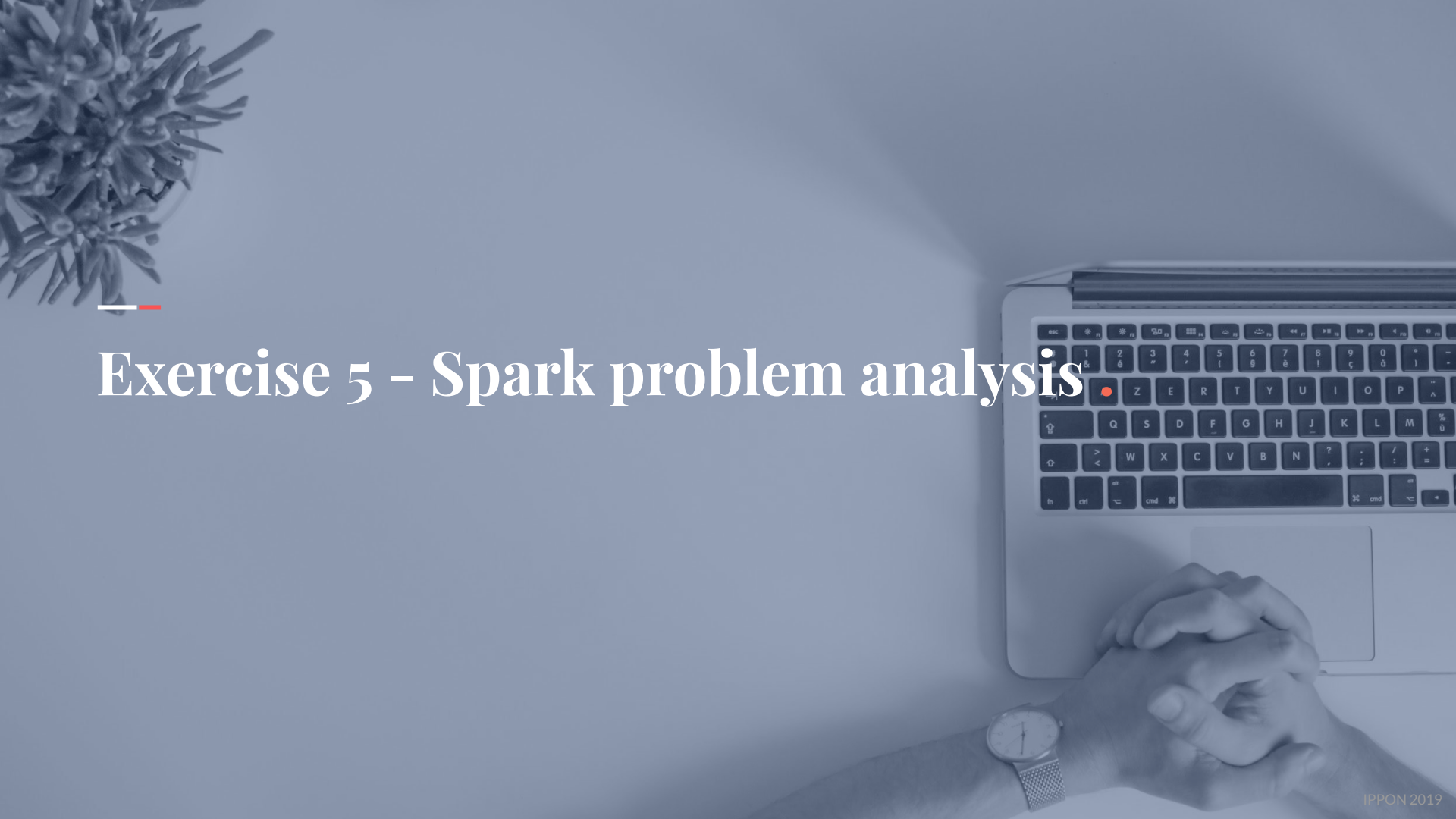
Index ▲	ID	Attempt	Status	Locality Level	Executor ID	Host	Launch Time	Duration	GC Time	Input Size / Records	Write Time	Shuffle Write Size / Records	Errors
0	1	0	SUCCESS	PROCESS_LOCAL	driver	localhost	2019/06/04 14:53:44	2 s	0,2 s	391.3 KB / 2824238	14 ms	42.9 KB / 3597	
1	2	0	SUCCESS	PROCESS_LOCAL	driver	localhost	2019/06/04 14:53:44	2 s	0,2 s	392.1 KB / 2822980	15 ms	25.9 KB / 2085	
2	3	0	SUCCESS	PROCESS_LOCAL	driver	localhost	2019/06/04 14:53:44	2 s	0,2 s	391 [illegible] / 2823209	74 ms	27 [illegible] / 2143	

Size of partition

Data exchanged

- **Rule of thumb:**
  - **2 - 3 partitions per cpu core**
  - **One partition :** 128Mb (block size HDFS)
  - **Number of partitions % CPUs allocated = 0**

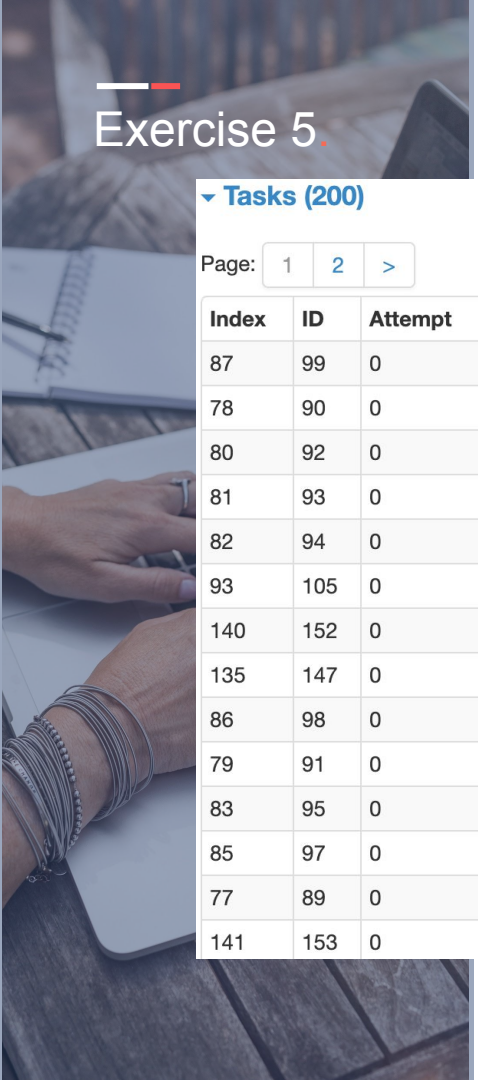
Exercise 5 - Spark problem analysis



Exercise 5 - Use case .

- Your Project Owner wants to add in his daily report, the IRIS income
- A coworker already maded the Spark program, this program does a join between the housing files and the income files
- However, it has a weird behaviour… Run the exercice 5 and try figure what is happening

Exercise 5.



▾ **Tasks (200)**

Page: 1 2 >

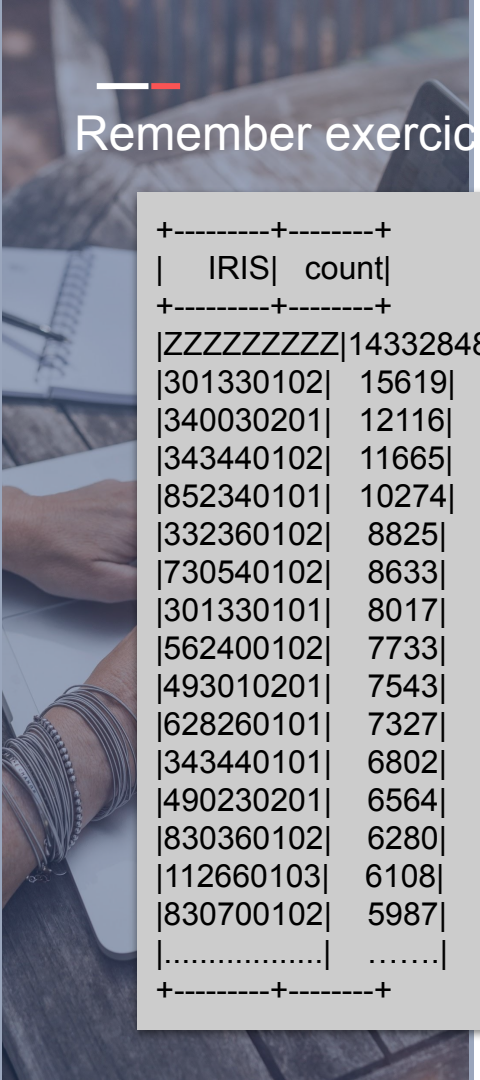
Index	ID	Attempt	Status	Locality Level	Executor ID	Host	Launch Time	Duration	GC Time
87	99	0	SUCCESS	ANY	driver	localhost	2019/06/03 15:39:41	6,1 min	7 s
78	90	0	SUCCESS	ANY	driver	localhost	2019/06/03 15:39:33	9 s	0,2 s
80	92	0	SUCCESS	ANY	driver	localhost	2019/06/03 15:39:33	8 s	0,1 s
81	93	0	SUCCESS	ANY	driver	localhost	2019/06/03 15:39:36	8 s	0,6 s
82	94	0	SUCCESS	ANY	driver	localhost	2019/06/03 15:39:37	7 s	0,6 s
93	105	0	SUCCESS	ANY	driver	localhost	2019/06/03 15:39:45	7 s	0,1 s
140	152	0	SUCCESS	ANY	driver	localhost	2019/06/03 15:40:20	7 s	0,4 s
135	147	0	SUCCESS	ANY	driver	localhost	2019/06/03 15:40:18	7 s	0,1 s
86	98	0	SUCCESS	ANY	driver	localhost	2019/06/03 15:39:40	7 s	0,6 s
79	91	0	SUCCESS	ANY	driver	localhost	2019/06/03 15:39:33	7 s	0,1 s
83	95	0	SUCCESS	ANY	driver	localhost	2019/06/03 15:39:37	7 s	0,6 s
85	97	0	SUCCESS	ANY	driver	localhost	2019/06/03 15:39:38	7 s	0,6 s
77	89	0	SUCCESS	ANY	driver	localhost	2019/06/03 15:39:32	7 s	0,1 s
141	153	0	SUCCESS	ANY	driver	localhost	2019/06/03 15:40:21	7 s	0,4 s

Exercise 5 - Step back.

Lets take a step back

- Our program joins:
  - IRIS housings
  - IRIS incomes

- Our join key is the IRIS column



Remember exercic

```
+---------+--------+
|     IRIS|   count|
+---------+--------+
|ZZZZZZZZZ|14332848|
|301330102|   15619|
|340030201|   12116|
|343440102|   11665|
|852340101|   10274|
|332360102|    8825|
|730540102|    8633|
|301330101|    8017|
|562400102|    7733|
|493010201|    7543|
|628260101|    7327|
|343440101|    6802|
|490230201|    6564|
|830360102|    6280|
|112660103|    6108|
|830700102|    5987|
|.....    |    .....|
+---------+--------+
```

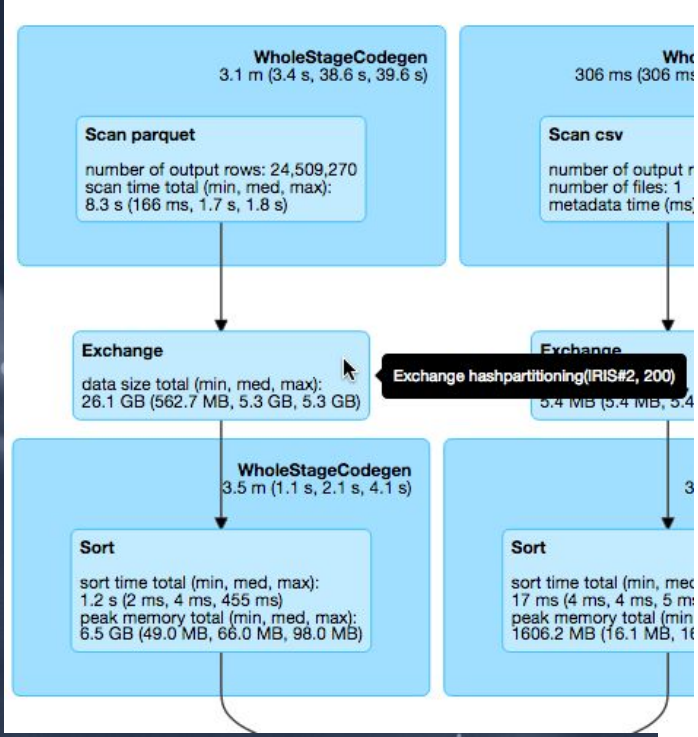
More than half of our keys contains ZZZZZZZZZ

Spark is smart … not too.

Index	ID	Attempt	Status	Locality Level	Executor ID	Host	Launch Time	Duration	GC Time	Output Size / Records	Shuffle Read Size / Records	Shuffle Spill (Memory)	Shuffle Spill (Disk)
87	99	0	SUCCESS	ANY	driver	localhost	2019/06/03 15:39:41	6,1 min	7 s	244.1 MB / 14384855	1775.3 MB / 14384923	14.0 GB	1639.3 MB
78	90	0	SUCCESS	ANY	driver	localhost	2019/06/03 15:39:33	9 s	0,2 s	1052.7 KB / 60129	7.5 MB / 60188	0.0 B	0.0 B
80	92	0	SUCCESS	ANY	driver	localhost	2019/06/03 15:39:33	8 s	0,1 s	926.8 KB / 48204	6.4 MB / 48275	0.0 B	0.0 B
81	93	0	SUCCESS	ANY	driver	localhost	2019/06/03 15:39:36	8 s	0,6 s	1094.1 KB / 59052	7.8 MB / 59106	0.0 B	0.0 B
82	94	0	SUCCESS	ANY	driver	localhost	2019/06/03 15:39:37	7 s	0,6 s	1009.5 KB / 54279	7.3 MB / 54352	0.0 B	0.0 B
93	105	0	SUCCESS	ANY	driver	localhost	2019/06/03 15:39:45	7 s	0,1 s	1171.8 KB / 63139	8.4 MB / 63211	0.0 B	0.0 B
140	152	0	SUCCESS	ANY	driver	localhost	2019/06/03 15:40:20	7 s	0,4 s	1015.7 KB / 61758	7.2 MB / 61819	0.0 B	0.0 B
135	147	0	SUCCESS	ANY	driver	localhost	2019/06/03 15:40:18	7 s	0,1 s	982.6 KB / 62385	7.2 MB / 62451	0.0 B	0.0 B
86	98	0	SUCCESS	ANY	driver	localhost	2019/06/03 15:39:40	7 s	0,6 s	1062.2 KB / 57668	7.6 MB / 57725	0.0 B	0.0 B
79	91	0	SUCCESS	ANY	driver	localhost	2019/06/03 15:39:33	7 s	0,1 s	843.6 KB / 44379	5.9 MB / 44437	0.0 B	0.0 B
83	95	0	SUCCESS	ANY	driver	localhost	2019/06/03 15:39:37	7 s	0,6 s	906.3 KB / 48250	6.4 MB / 48309	0.0 B	0.0 B
85	97	0	SUCCESS	ANY	driver	localhost	2019/06/03 15:39:38	7 s	0,6 s	913.6 KB / 48514	6.5 MB / 48579	0.0 B	0.0 B
77	89	0	SUCCESS	ANY	driver	localhost	2019/06/03 15:39:32	7 s	0,1 s	966.1 KB / 51996	7.0 MB / 52062	0.0 B	0.0 B
141	153	0	SUCCESS	ANY	driver	localhost	2019/06/03 15:40:21	7 s	0,4 s	1039.2 KB / 57585	7.4 MB / 57658	0.0 B	0.0 B
142	154	0	SUCCESS	ANY	driver	localhost	2019/06/03 15:40:24	6 s	0,3 s	999.2 KB / 53468	7.0 MB / 53533	0.0 B	0.0 B

Spark put all the ZZZ entries in a single partition

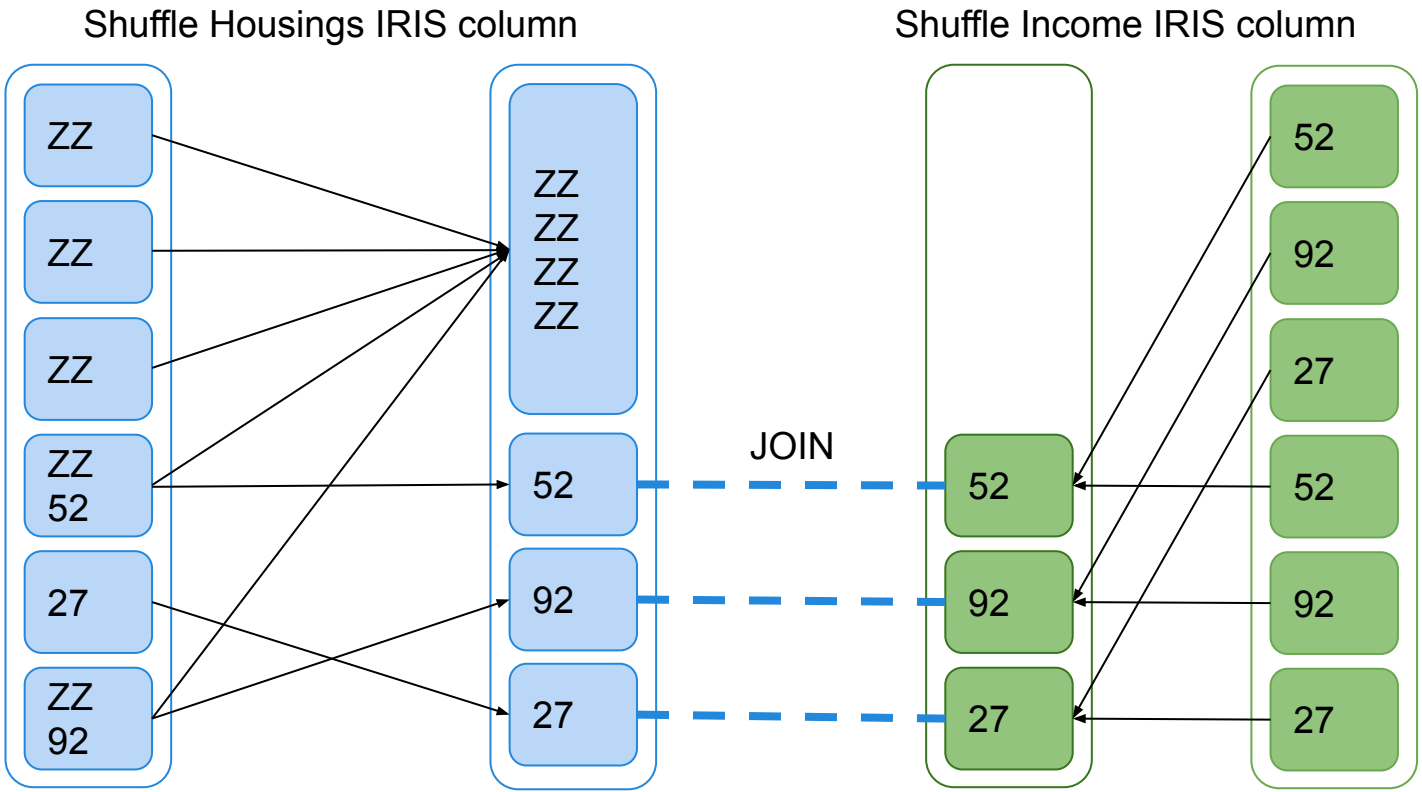
Exercise 5 - Skew.



This data distribution problem is called Skew

- ★ Before a shuffle, Spark use the HashPartitioner to partition data
  - Calculate a Hash from all the keys (Object.hashcode())
  - Put data with same keys in same partitions

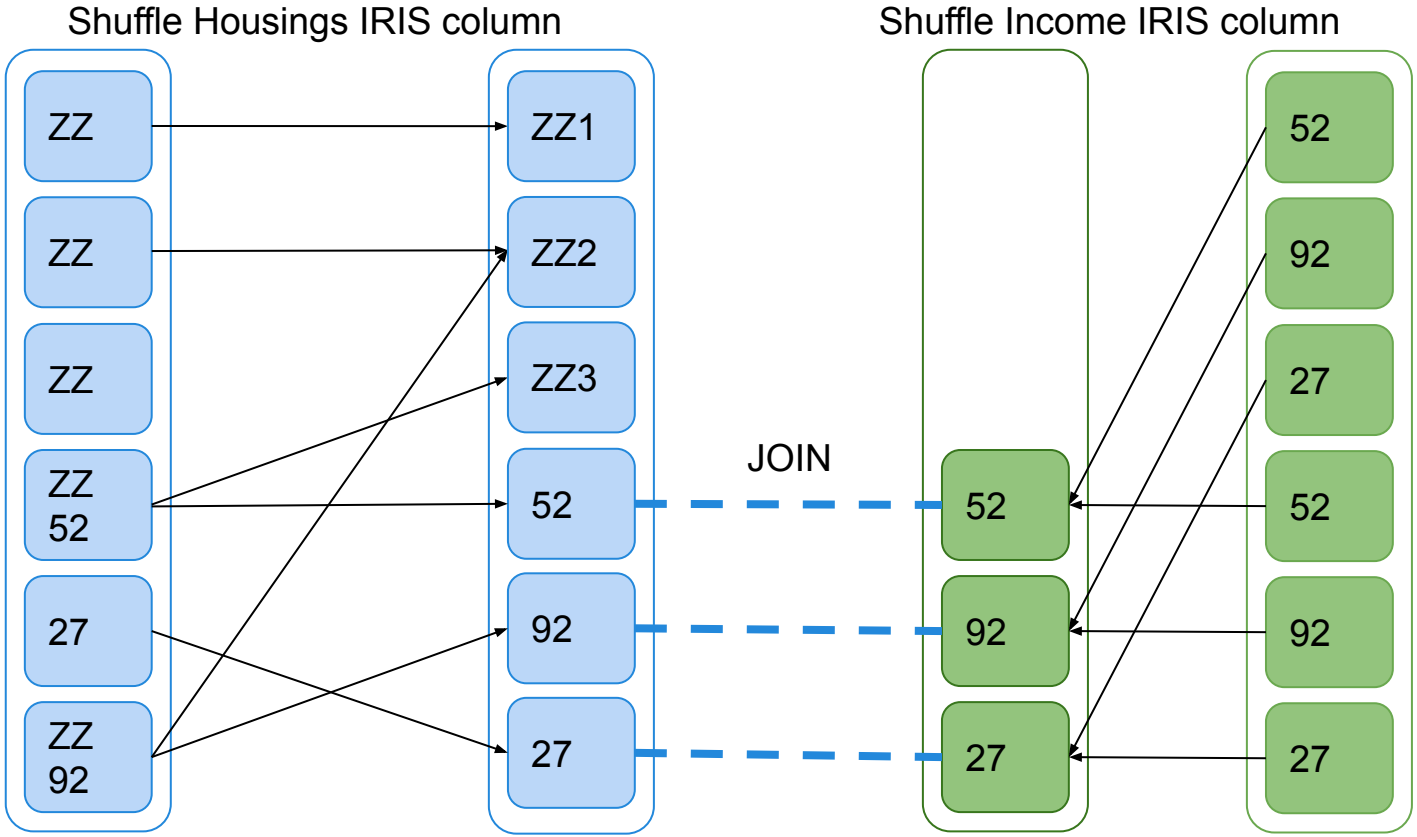
Exercice 5 : Exam



Exercise 5 - Step back.

- To correct this, we need to perform a special technique called “Salting”
  - Add a random number each time our key contains ZZZZZZZZ

Exercice 5 : Correc



Exercise 5 - Solution.

In our initial DataFrame

- ★ Create an IRIS_BACKUP wich is a copy of the IRIS column (df.withColumn)
- ★ Create a CASE WHEN condition on the IRIS column
  - use concat() to merge multiple Strings
  - use lit(“string”) to transform a String to a column
  - rand() generates a random number between 0 and 1

```
df.withColumn("xxxx",
  when(col("xxxx") === saltedValue,doSomething).otherwise(col("IRIS"))
)
```

Exercise 5 - Conclusion.

Know your data!

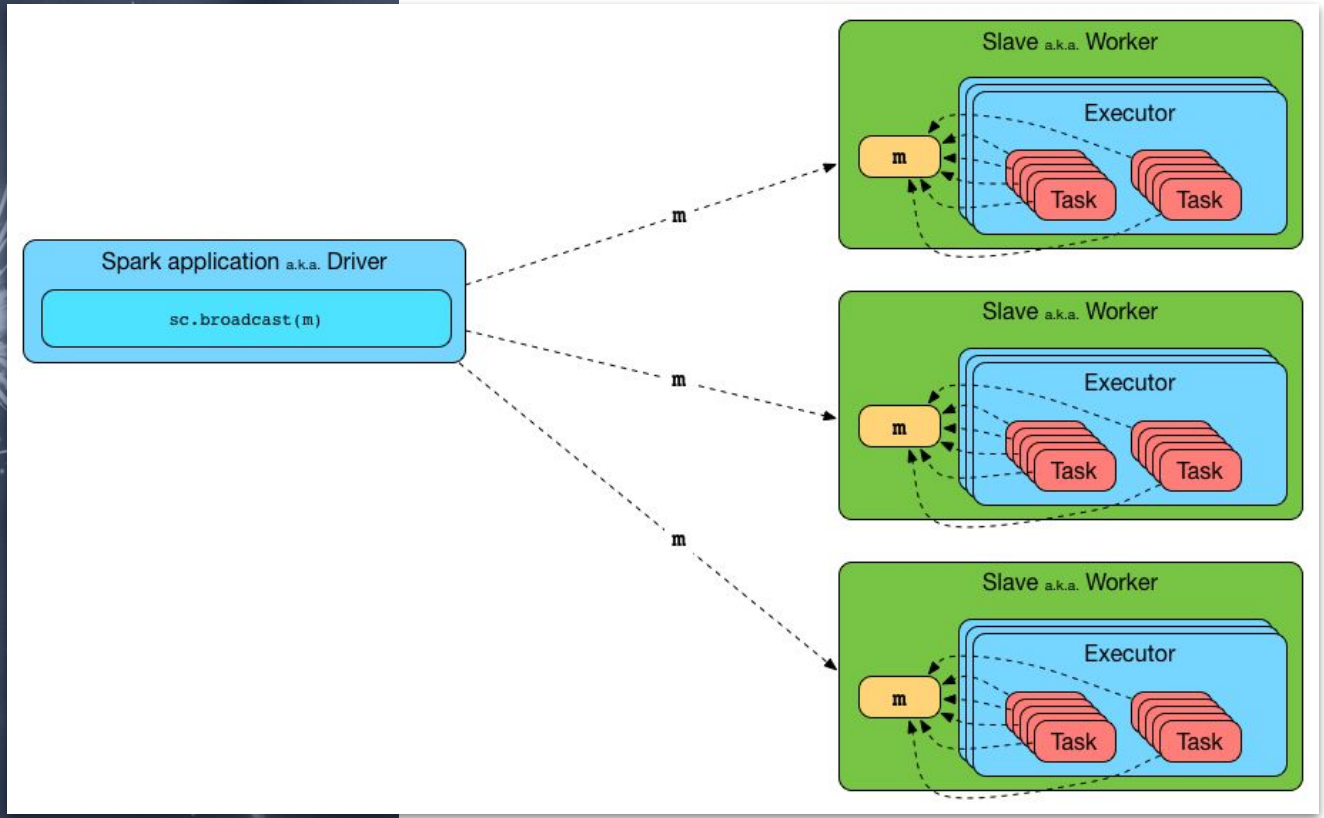
- ★ Everytime a partition take way more time than others to perform join actions, you are dealing with skewed data
- ★ Don’t “oversalt” your data, it might produce too much partitions as a result

Exercise 6 - Joins .

Exercise 6 - Use case.

- You can avoid shuffling by broadcasting “small enough” datasets
- We are joining a 1.3G shuffle for a 500Mo file
- Our other dataset is actually quite small
  - Implement the join by broadcasting the iris dataset

Exercise 6 - Broadcast example.



Exercise 6 - Solution.

There are multiple ways to broadcast variables

1. Create a Map who contains your data, then broadcast it
    - Create an udf who performs the computation
    - Call your udf inside a .withColumn

2. Use the classic DataFrame functions

logements.join(broadcast(iris), Seq("keys"), "type")

3. Spark automaticly broadcast DataFrames if they are under 10M
    - spark.sql.autoBroadcastJoinThreshhold

    This function is disabled in this exercise :D

Exercise 6 - Small enough?.

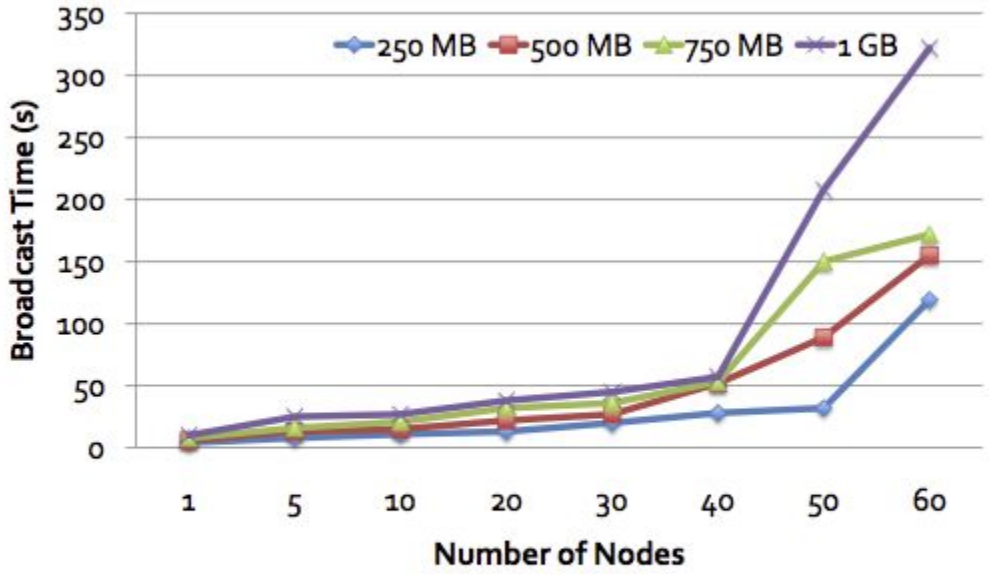


Figure 5: Scalability and performance of CHB (All nodes: m1.large EC2 instances).