BigData Spark – Hands-On

Optimisations

SQL Execution Plan, DAG, SparkUI
Performances

Arnaud Nauwynck

Dec 2023

Objectives of Hands-On

Reminders:
Local Install, spark-shell
Spark File IO
MetaStore tables

1/ Execution Plan, SQL Explain / dataset.explain

2/ Spark UI, DAG, Narrow/Wide Transformations

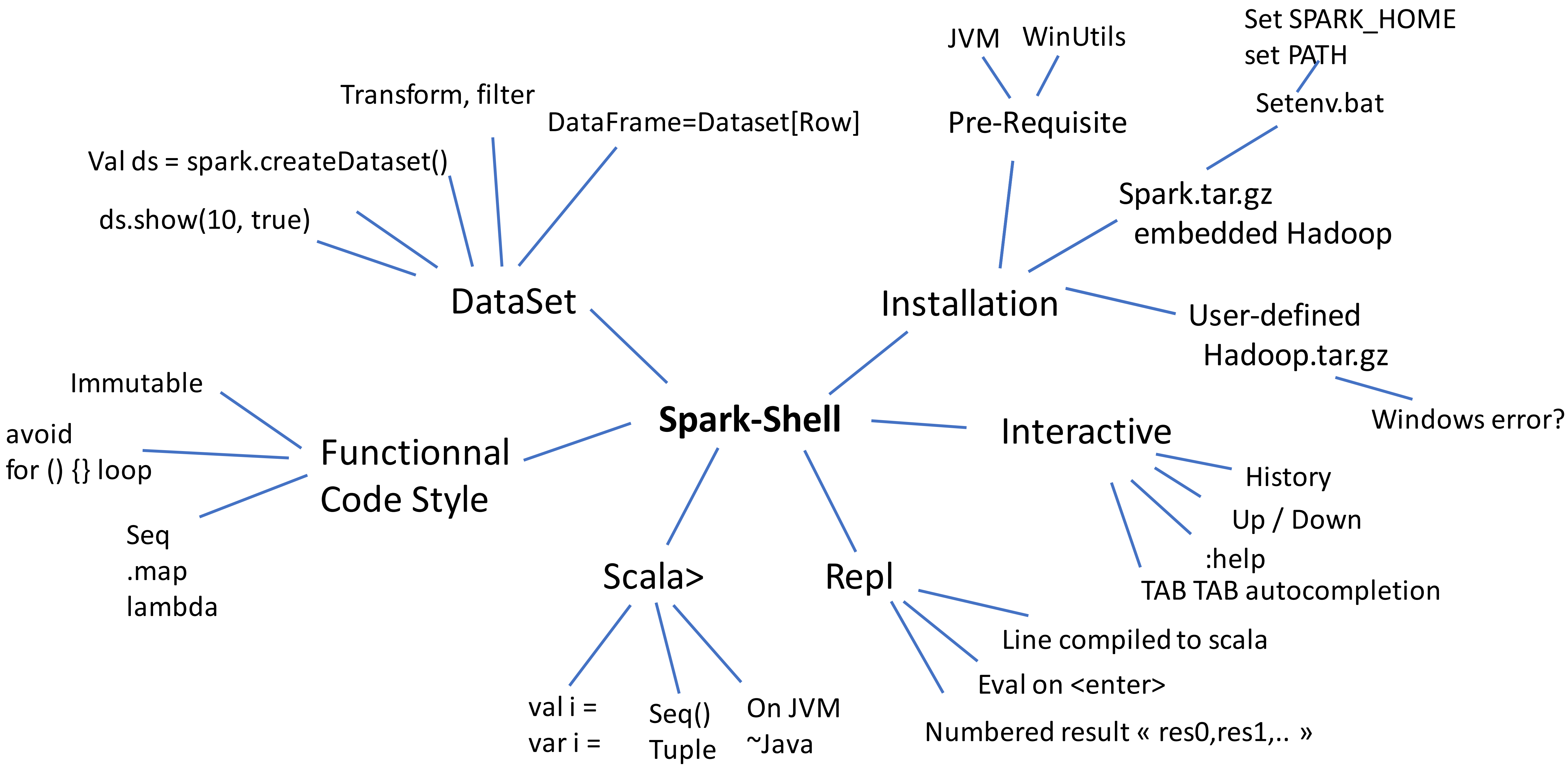
3/ RDD cache / checkpoint

4/ Partition Pruning / Columns Pruning / Predicate-Push-Down

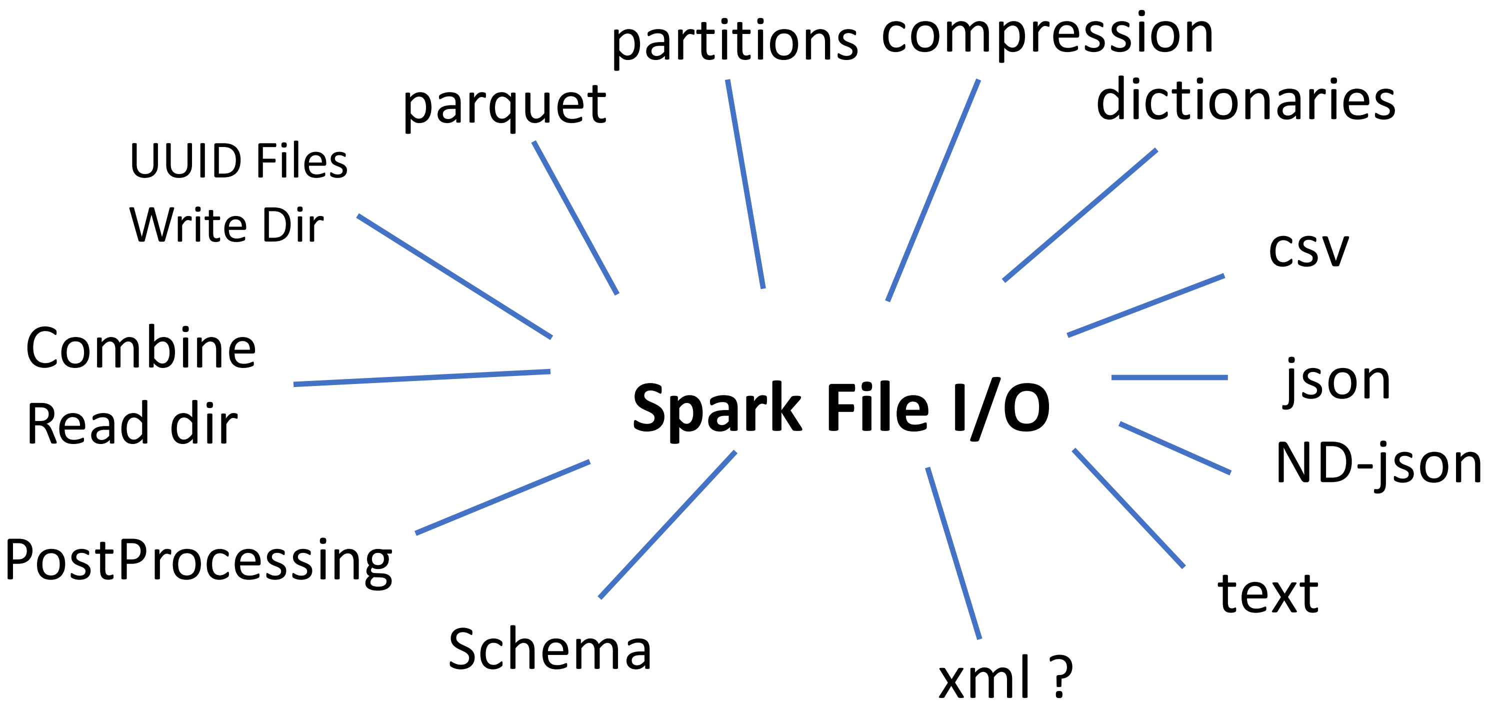
5/ Parquet Optims (Sort, Stats, Dictionary, Bloom, BlockSize)

6/ perfs: SpillToDisk, JOINs, hint

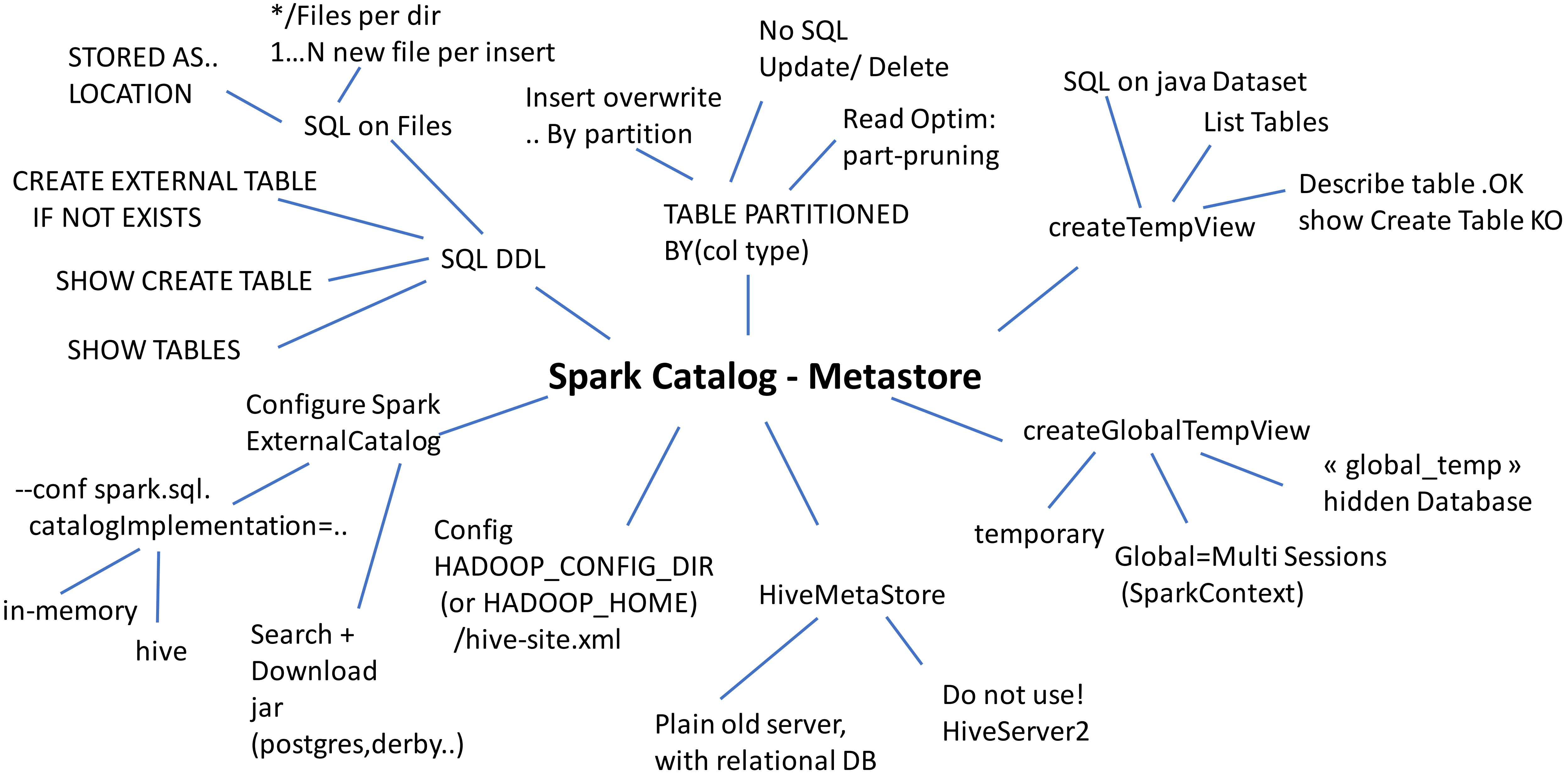
Reminder Hands-On 1



Reminder Hands-On 2



Reminder Hands-On 3



Objectives of Hands-On

1/ Discover Spark UI, Execution Plan, Job - Stage

2/ Understand DAG, Narrow/Wide Transformations

3/ RDD cache / checkpoint

4/ Partition Pruning / Columns Pruning / Predicate-Push-Down

5/ Parquet Optims (Sort, Stats, Dictionary, Bloom, BlockSize)

6/ perfs: SpillToDisk, JOINs, hint

Pre-Requisites

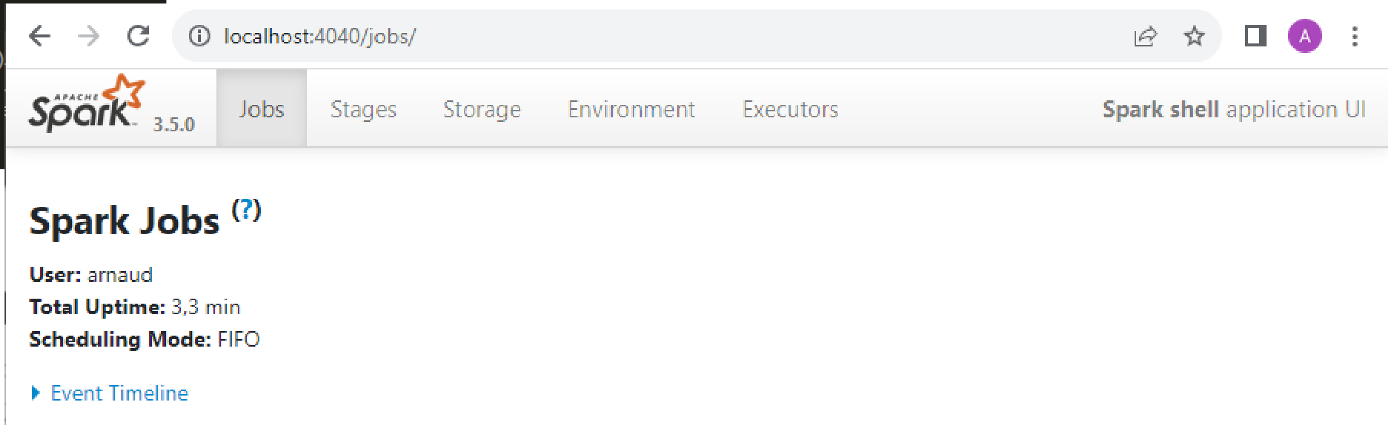
1/ Launch spark-shell **spark-shell --driver-memory 8g**

2/ Open Web UI: **http://localhost:4040**

```
C:\data>spark-shell --driver-memory 8g
Setting default log level to "WARN".
To adjust logging level use sc.setLogLevel(newLevel). For SparkR, use setLogLevel(newLevel).
Welcome to
      ____              __
     / __/__  ___ _____/ /__
    _\ \/ _ \/ _ `/ __/  '_/
   /___/ .__/\_,_/_/ /_/\_\   version 3.5.0
      /_/

Using Scala version 2.13.8 (OpenJDK 64-Bit Server VM, Java 20.0.1)
Type in expressions to have them evaluated.
Type :help for more information.
Spark context Web UI available at http://DesktopArnaud:4040
Spark context available as 'sc' (master = local[*], app id = local-1701612093505)
Spark session available as 'spark'.

scala>
```

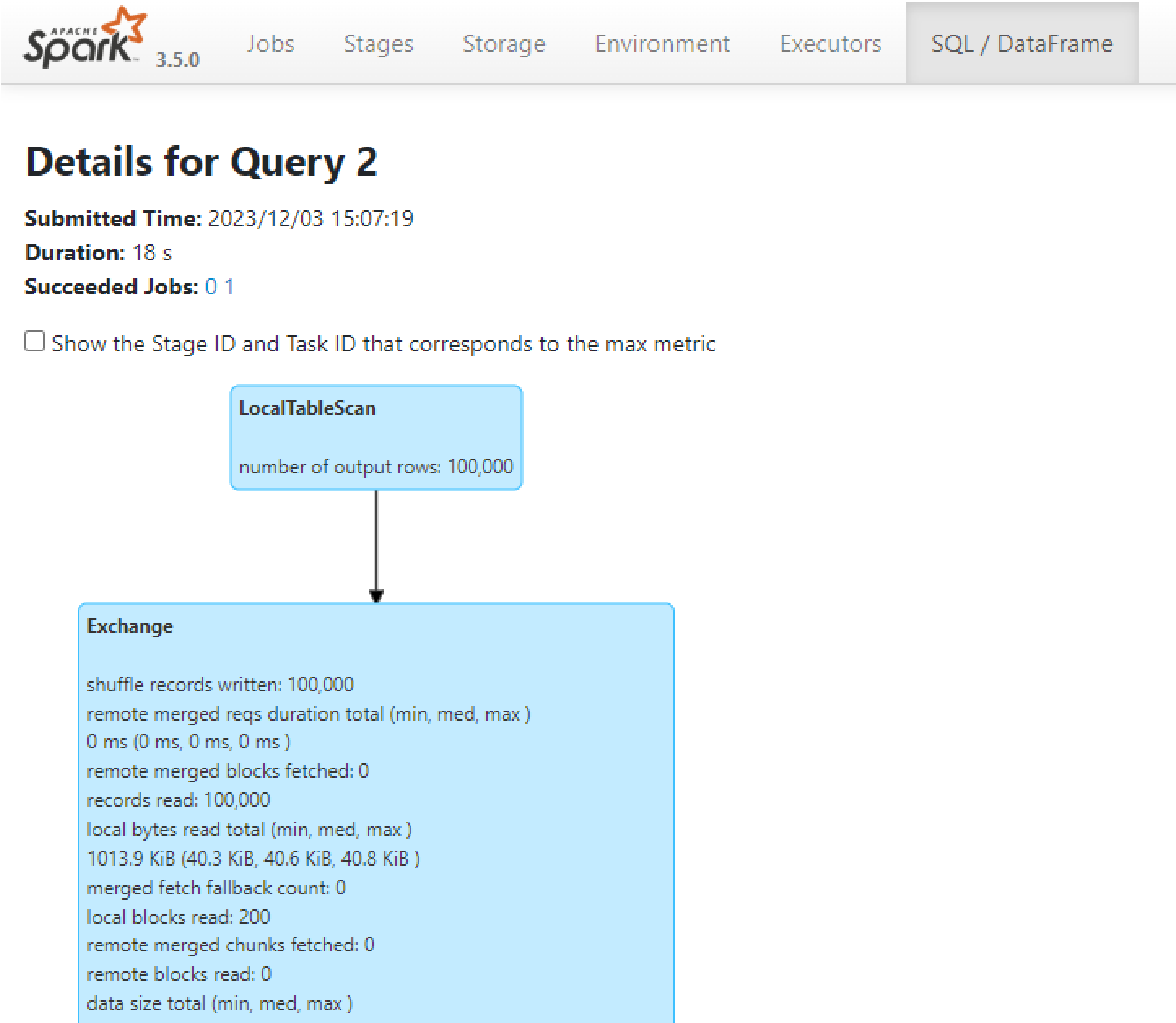


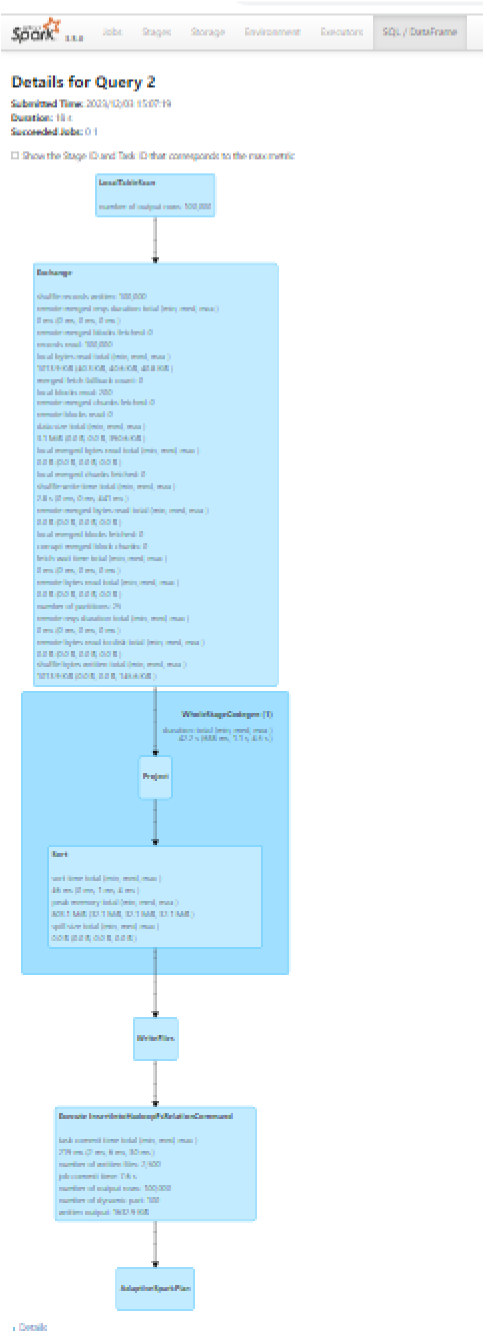
Exercise 1: create Table - repartition- insertInto

```
val ds = spark.createDataset((1 to 1000000).map(x => (s"$x", x % 100)))

// TEST_PART1:
spark.sql("drop table if exists db1.test_part1");
spark.sql("create table db1.test_part1 (col1 string) partitioned by (part1 int) stored as parquet");
ds.repartition(25).write.mode("overwrite").insertInto("db1.test_part1");
```

Refresh Spark UI
explore all **SQL(1/3)** > Job > Stages





Refresh Spark UI
explore all tabs SQL > **Jobs(2)** > Stages

Spark 3.5.0 | Jobs | Stages | Storage | Environment | Executors | SQL / DataFrame | **Spark shell** application UI

Spark Jobs (?)

User: arnaud
Total Uptime: 6,1 min
Scheduling Mode: FIFO
Completed Jobs: 2

▸ Event Timeline

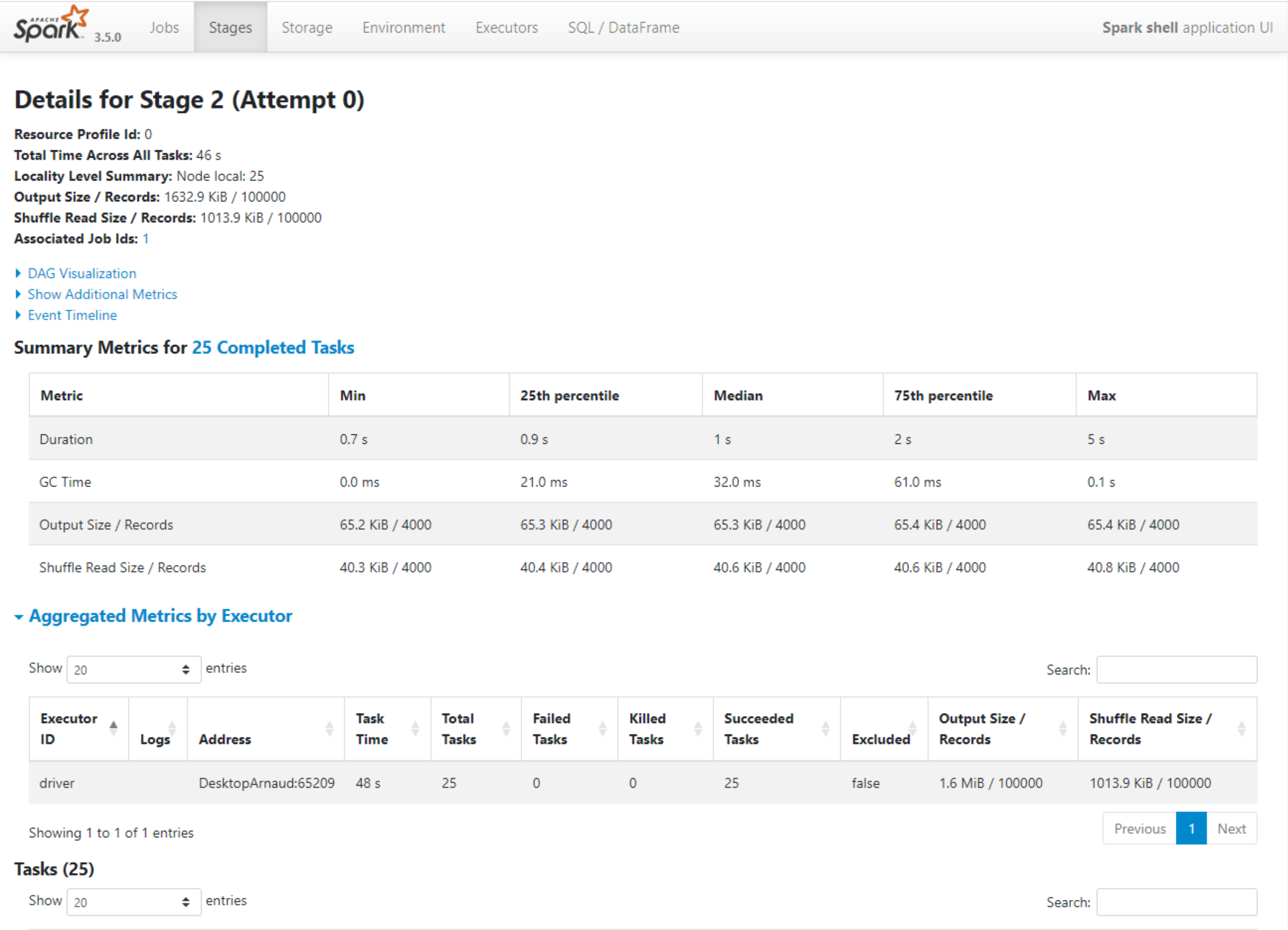
▾ Completed Jobs (2)

Page: 1 — 1 Pages. Jump to 1 . Show 100 items in a page. Go

Job Id ▾	Description	Submitted	Duration	Stages: Succeeded/Total	Tasks (for all stages): Succeeded/Total
1	insertInto at <console>:1 insertInto at <console>:1	2023/12/03 15:07:21	6 s	1/1 (1 skipped)	25/25 (8 skipped)
0	insertInto at <console>:1 insertInto at <console>:1	2023/12/03 15:07:19	1 s	1/1	8/8

Page: 1 — 1 Pages. Jump to 1 . Show 100 items in a page. Go

Refresh Spark UI
explore all SQL > Jobs > **Stages(3/3)**



Exercise 1: … Analysis

1/ explain the directory and files structures of result table

2/ Describe the "insert" action

3/ Explain how Spark executes the action
(how many distinct partition values?
how RDD partitions are divided?
so how many distinct partition values per RDD partition ?
how many jobs / stages / tasks, and side effect of each task ?)

4/ explain (and check on your disk) the number of parquet files written

5/ justify with Spark UI screenshots

Exercise 2: Similar insert, different table

Similar insert, but .repartition(..) by column "_2", instead of .repartition(25)

```
spark.sql("drop table if exists db1.test_part2");
spark.sql("create table db1.test_part2 (col1 string) partitioned by (part1 int) stored as parquet");
ds.repartition($"_2").write.mode("overwrite").insertInto("db1.test_part2");
```

Exercise 2: Analysis

1/ explain the directory and files structures of result table

2/ Describe the "insert" action

3/ Explain how Spark executes the action
(how many distinct partition column values?
how RDD partitions are divided?
so how many distinct partition values per RDD partition ?
how many jobs / stages / tasks, and side effect of each task ?)

4/ explain (and check on your disk) the number of parquet files written

5/ justify with Spark UI screenshots

Exercise 3: Similar insert, different table

Similar insert, but .repartition(..) by column "_1", instead of .repartition(25)
emulate an older version of Spark(<3.2 = without the Adaptive Query Execution)

```
spark.conf.set("spark.sql.adaptive.enabled", "false");
spark.sql("drop table if exists db1.test_part3");
spark.sql("create table db1.test_part3 (col1 string) partitioned by (part1 int) stored as parquet");
ds.repartition($"_1").write.mode("overwrite").insertInto("db1.test_part3");
spark.conf.set("spark.sql.adaptive.enabled", "true")
```

Exercise 3: Analysis

1/ explain the directory and files structures of result table

2/ Describe the "insert" action

3/ Explain how Spark executes the action
(how many distinct partition column values?
how RDD partitions are divided?
so how many distinct partition values per RDD partition ?
how many jobs / stages / tasks, and side effect of each task ?)

4/ explain (and check on your disk) the number of parquet files written

5/ justify with Spark UI screenshots

Exercise 4: Similar insert, different table (Advanced: Adaptive Query Execution !)

Similar insert, but .repartition(..) by column "_1"
using a recent version of Spark (>=3.2 = with AQE optim)

```
spark.sql("drop table if exists db1.test_part4");
spark.sql("create table db1.test_part4 (col1 string) partitioned by (part1 int) stored as parquet");
ds.repartition($"_1").write.mode("overwrite").insertInto("db1.test_part4");
```

Exercise 4: Analysis

1/ explain the directory and files structures of result table

2/ Describe the "insert" action

3/ Explain how Spark executes the action
(how many distinct partition column values?
how partitions are divided?
What did Spark to detect & fix abnormal usage of partitions like in Exercise 3/ ?
how many jobs / stages / tasks, and side effect of each task ?)

4/ explain (and check on your disk) the number of parquet files written

5/ justify with Spark UI screenshots

Exercise 5

For execution of exercise 4/, compare and explain

a/ the spark API code to create linked Dataset objects that depends of each others

b/ the DAG view of the Logical Plan in Spark UI tab "Sql"

c/ the ascii printed version of the Plan (Logical or Physical)
you can obtain it by clicking on ▸ Details on bottom of SQL tab

d/ the DAG view of Job-Stage in Spark UI tab "Jobs"

Do drawings explaining how Spark goes from a/ to b/ (or c/), then from b/ to d/

Let's switch to use Spark UI for "SELECT" queries

We will study the optimization Plan on File reads
(Partition pruning, Predicate-Push-Down)

We will reuse the dataset (and tables) on the OpenData.gouv "BAL addresses"

Exercise 6: Query Partitioned Table WHERE partitionColumn=..

a/ remind on previous Hands-On TD3

SHOW CREATE TABLE db1.address => .. Un-partitioned table, PARQUET

SHOW CREATE TABLE db1.address_by_dept => .. Partitioned table, PARQUET

b/ execute queries

```
SELECT * FROM address_by_dept WHERE dept=92
SELECT * FROM address_by_dept WHERE dept in (75,78,91,92)
SELECT * FROM address_by_dept WHERE commune_name = ‘Nanterre’
```

c/ Remind which dir / files should be read by spark… which query is faster

Exercise 7: Execute « EXPLAIN <<SQL>> »

a/ Execute SQL… prefixed by “EXPLAIN” keyword

spark.sql("**EXPLAIN** select count(*) FROM db1.address_by_dept WHERE dept=92")

Hint: display nicely, using .show(false) OR .foreach(println(_))

b/ do you see “PartitionFilters: [..]” ?

Exercise 7(next): using dataset.explain() API

```
val ds = spark.sql("select count(*) FROM db1.address_by_dept WHERE dept=92")
ds.explain

ds.explain(false)

ds.explain(true)
```

Exercise 8 : compare « EXPLAIN » queries

a/ Compare both

spark.sql("**EXPLAIN** select count(*) FROM db1.address_by_dept WHERE dept=92")

spark.sql("**EXPLAIN** select count(*) FROM db1.address WHERE dept=92")

b/ what changed ?

Exercise 9 : dataset parralelism (N partitions) => show N tasks in Job->Stage->Tasks

a/ browse from SQL logical plan to execution plan: Jobs->Stages

b/ take SparkUI screenshot showing detailed table results
(time, IO statistics) PER partition.

c/ Check table results has N rows,
where N=ds.toJavaRDD.getNumPartition
Is the dataset well balanced (or skewed) ?

d/ take screenshot of (min-average-max) partition results in DAG

Objectives of Hands-On

1/ Execution Plan, SQL Explain / dataset.explain

2/ Spark UI, DAG,
Narrow/Wide Transformations

3/ RDD cache / checkpoint

4/ Partition Pruning / Columns Pruning / Predicate-Push-Down

5/ Parquet Optims (Sort, Stats, Dictionary, Bloom, BlockSize)

6/ perfs: SpillToDisk, JOINs, hint

Reminder: Narrow/Wide Transformations

https://spark.apache.org/docs/latest/rdd-programming-guide.html#transformations

Transformations

The following table lists some of the common transformations supported by Spark. Refer to the RDD API doc (Scala, Java, Python, R) and pair RDD functions doc (Scala, Java) for details.

Transformation	Meaning
map(*func*)	Return a new distributed dataset formed by passing each element of the source through a function *func*.
filter(*func*)	Return a new dataset formed by selecting those elements of the source on which *func* returns true.
flatMap(*func*)	Similar to map, but each input item can be mapped to 0 or more output items (so *func* should return a Seq rather than a single item).
mapPartitions(*func*)	Similar to map, but runs separately on each partition (block) of the RDD, so *func* must be of type Iterator<T> => Iterator<U> when running on an RDD of type T.
mapPartitionsWithIndex(*func*)	Similar to mapPartitions, but also provides *func* with an integer value representing the index of the partition, so *func* must be of type (Int, Iterator<T>) => Iterator<U> when running on an RDD of type T.
sample(*withReplacement*, *fraction*, *seed*)	Sample a fraction *fraction* of the data, with or without replacement, using a given random number generator seed.
union(*otherDataset*)	Return a new dataset that contains the union of the elements in the source dataset and the argument.
intersection(*otherDataset*)	Return a new RDD that contains the intersection of elements in the source dataset and the argument.
distinct([*numPartitions*]))	Return a new dataset that contains the distinct elements of the source dataset.
groupByKey([*numPartitions*])	When called on a dataset of (K, V) pairs, returns a dataset of (K, Iterable<V>) pairs. **Note:** If you are grouping in order to perform an aggregation (such as a sum or average) over each key, using `reduceByKey` or `aggregateByKey` will yield much better performance. **Note:** By default, the level of parallelism in the output depends on the number of partitions of the parent RDD. You can pass an optional `numPartitions` argument to set a different number of tasks.
reduceByKey(*func*, [*numPartitions*])	When called on a dataset of (K, V) pairs, returns a dataset of (K, V) pairs where the values for each key are aggregated using the given reduce function *func*, which must be of type (V,V) => V. Like in `groupByKey`, the number of reduce tasks is configurable through an optional second argument.
aggregateByKey(*zeroValue*)(*seqOp*, *combOp*, [*numPartitions*])	When called on a dataset of (K, V) pairs, returns a dataset of (K, U) pairs where the values for each key are aggregated using the given combine functions and a neutral "zero" value. Allows an aggregated value type that is different than the input value type, while avoiding unnecessary allocations. Like in `groupByKey`, the number of reduce tasks is configurable through an optional second argument.
sortByKey([*ascending*], [*numPartitions*])	When called on a dataset of (K, V) pairs where K implements Ordered, returns a dataset of (K, V) pairs sorted by keys in ascending or descending order, as specified in the boolean `ascending` argument.
join(*otherDataset*, [*numPartitions*])	When called on datasets of type (K, V) and (K, W), returns a dataset of (K, (V, W)) pairs with all pairs of elements for each key. Outer joins are supported through `leftOuterJoin`, `rightOuterJoin`, and `fullOuterJoin`.
cogroup(*otherDataset*, [*numPartitions*])	When called on datasets of type (K, V) and (K, W), returns a dataset of (K, (Iterable<V>, Iterable<W>)) tuples. This operation is also called `groupWith`.
cartesian(*otherDataset*)	When called on datasets of types T and U, returns a dataset of (T, U) pairs (all pairs of elements).
pipe(*command*, *[envVars]*)	Pipe each partition of the RDD through a shell command, e.g. a Perl or bash script. RDD elements are written to the process's stdin and lines output to its stdout are returned as an RDD of strings.
coalesce(*numPartitions*)	Decrease the number of partitions in the RDD to numPartitions. Useful for running operations more efficiently after filtering down a large dataset.
repartition(*numPartitions*)	Reshuffle the data in the RDD randomly to create either more or fewer partitions and balance it across them. This always shuffles all data over the network.
repartitionAndSortWithinPartitions(*partitioner*)	Repartition the RDD according to the given partitioner and, within each resulting partition, sort records by their keys. This is more efficient than calling `repartition` and then sorting within each partition because it can push the sorting down into the shuffle machinery.

Exercise 10: combine several **Narrow** transformations filter, sample, select, withColumn …

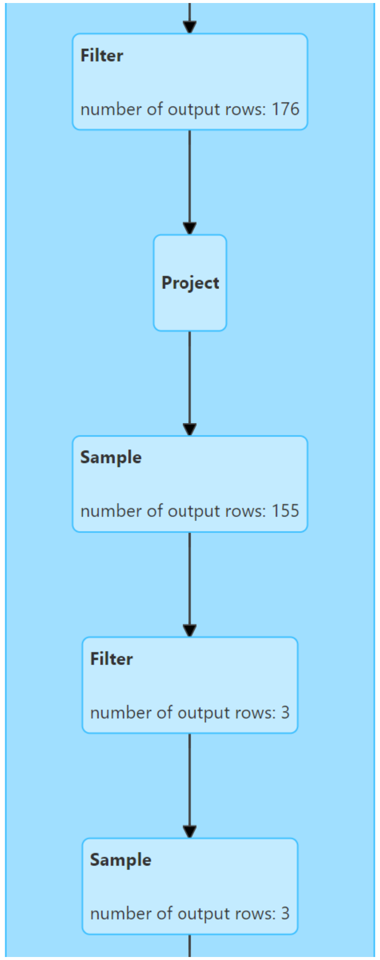
a/ Execute query like

```
 val addressDs = …..
  .filter(a).filter(b)   // => check Spark combine as   .filter(a && b) !!
```

b/ ..like

```
addressDs
     .withColumn(..)
     .filter( .. ) .sample(0.9)
     .filter( .. ) .sample (0.9)
     .count
```

b/ Check in Spark UI that there is only 1 « **WholeStageCodegen (1)** » containing several instructions



Exercise 11: WholeStageCodegen java code for(;;) on Dataset<Row>…

a/ Show the corresponding generated Java Code of WholeStageCodegen

use **ds.queryExecution.debug.codegen**

b/ read it …
find « .. extends org.apache.spark.sql.execution.BufferedRowIterator »
https://github.com/apache/spark/blob/master/sql/core/src/main/java/org/apache/spark/sql/execution/BufferedRowIterator.java#L97

c/ find method @Override **protected void processNext() {**
.. It is supposed to map copy all filtered rows x columns to new RDD[Row]

d/ do you find your filter conditions on your columns ?

Exercise 12: cascade several Wide Transformations: repartition, groupByKey, join, distinct..

a/ Execute query like

```
addressDs
.repartition(3)
.repartition(2, $"commune_insee")
.repartition(4)
.repartition(2, $"commune_nom")
.count
```

b/ Check in Spark UI that there are N shuffles

Exercise 13: Check on 2 consecutive Stages that previous Shuffle Write = next Shuffle Read

Completed Stages (4)

Page: 1

1 Pages. Jump to 1 . Show 100 items in a page. Go

Stage Id ▾	Description		Submitted	Duration	Tasks: Succeeded/Total	Input	Output	Shuffle Read	Shuffle Write
9	show at <console>:23	+details	2022/10/06 17:17:44	41 ms	1/1				
5	show at <console>:23	+details	2022/10/06 17:17:43	0,6 s	2/2			78.8 MiB	36.2 MiB
2	show at <console>:23	+details	2022/10/06 17:17:41	2 s	1/1			$\boxed{\text{39.6 MiB}}$	78.8 MiB
0	show at <console>:23	+details	2022/10/06 17:17:39	2 s	8/8	27.2 MiB			$\boxed{\text{39.6 MiB}}$

Exercise 14: mix cascade of Narrow and Wide

Example

```
spark.sql("select * from address3")
.repartition(1).sample(0.9)
.repartition(2).sample(0.8).filter("commune_nom like '%a%'")
.repartition(3).sample(0.7).filter("commune_nom like '%t%'").sample(0.6)
.count
```

a/ Explain how many Shuffles you expect

b/ How many instructions you expect in each

WholeStageCodegen (1), WholeStageCodegen (2), WholeStageCodegen (3)

Objectives of Hands-On

1/ Execution Plan, SQL Explain / dataset.explain

2/ Spark UI, DAG, Narrow/Wide Transformations

3/ RDD cache / checkpoint

4/ Partition Pruning / Columns Pruning / Predicate-Push-Down

5/ Parquet Optims (Sort, Stats, Dictionary, Bloom, BlockSize)

6/ perfs: SpillToDisk, JOINs, hint

Exercise 15: Several Actions on same Dataset

a/ Execute several actions on same Dataset
  ds.show
  ds.show

b/ equivalently… several SQL
  select * from address
  select * from address

b/ Check in SparkUI that ds is recomputed each time
  (show screenshots with job ids, stage ids)

c/ Check Spark File IO Statistics, Shuffle, Time elapsed

Exercise 16: Avoiding re-computation dataset .cache()

a/ Same as Exercise 12… but use before

ds.cache() // or equivalent: .persist()

Execute several actions on cached Dataset
  ds.show
  ds.show

b/ Check in SparkUI that ds is NOT recomputed
  … But Full lineage is still displayed several time in greyed / with green point
  give screenshots showing job ids / stage ids

c/ check in SparkUI > Storage > RDD

d/ and after … **ds.unpersist()**

Exercise 17: Execute complex DAG on dataset with lot of duplicates

Example: dataset repeated, with small variations, then union …

```
val ds1 = spark.sql("select * from address3").repartition(2).filter("commune_nom like
'%n%'").repartition(2).sample(0.9);
val ds2 = ds1.union(ds1).limit(1000).repartition(2).sample(0.9);
val ds3 = ds2.union(ds2).limit(1000).repartition(2).sample(0.9).union(ds2);
val ds4 = ds3.union(ds3).limit(1000).repartition(2).sample(0.9).union(ds3);
val ds5 = ds4.union(ds4).limit(1000).repartition(2).sample(0.9).union(ds3);
ds5.count;
```

Open corresponding DAG in Spark UI

Exercise 18: How to simplify DAG display ? … .checkpoint() !

a/ ensure
sc.setCheckpointDir(« c:/data/checkpoint-dir »)

b/ Same as Exercise 15… but use checkpoint:
ds3Before=..
val ds3 = ds3Before.checkpoint(); // IMPORTANT TO RE-ASSIGN and use new
ds4=..

c/ Check in SparkUI that ds3 is persisted,
… AND lineage no more displayed

d/ there is NO « unCheckpoint() » as there is for unpersist()

Objectives of Hands-On

1/ Execution Plan, SQL Explain / dataset.explain

2/ Spark UI, DAG, Narrow/Wide Transformations

3/ RDD cache / checkpoint

4/ Partition Pruning / Columns Pruning / Predicate-Push-Down

5/ Parquet Optims (Sort, Stats, Dictionary, Bloom, BlockSize)

6/ perfs: SpillToDisk, JOINs, hint

Exercise 19: Reminder Partition Pruning

There are 3 « Pruning » Optimizations done by Spark
1/ Column Pruning
2/ Partition Pruning
3/ Predicate-Push-Down

Questions: (interogation on Spark Lesson)

a/ explain in few words what is 1/, 2/, 3/

b/ what are pros/cons of many (small) partitions ?

Exercise 19: Column Pruning

Reminding that Parquet is a « Columnar File Format »

Check by reading only 1 column, that Spark does not read fully Parquet Files
… only Page block of selected column.
This is « Column Pruning »

Example:
spark.sql(« select distinct commune_nom from address »).count

Check in SparkUI the File Bytes read statistics.
Compare with total file size
Check in SparkUI the IO read statistics per partition tasks
Give screenshots of SparkUI

Exercise 20: Predicate-Push-Down

a/ Execute following Queries

select count(*) from db1.address where commune_nom = ‘Nanterre’
select count(*) from db1.address where UPPER(commune_nom) = ‘NANTERRE’
select count(*) from db1.address where commune_nom like ‘%Nanterre%’

b/ Check in SparkUI the execution Plan
Search for « PushedFilters:[..] »

c/ For which query Spark is able to push down « some/all conditions » to parquet library ?

d/ explain how parquet files should be saved for having good read performances

Hint: use sc.setCallSite(« query comment») to find easily your query in Spark UI

Objectives of Hands-On

1/ Execution Plan, SQL Explain / dataset.explain

2/ Spark UI, DAG, Narrow/Wide Transformations

3/ RDD cache / checkpoint

4/ Partition Pruning / Columns Pruning / Predicate-Push-Down

5/ Parquet Optims (Sort, Stats, Dictionary, Bloom, BlockSize)

6/ perfs: SpillToDisk, JOINs, hint

Exercise 20: prepare sorted parquet table for Predicate-Push-Down

We want to have few PARQUET files (1 or several, but not hundred)
Each PARQUET File split in several blocks of 16Mo (default=256Mo)
Each block sorted to have efficient Dictionaries and Min-Max Statistics

```
val allAdressCsvDs = spark.read.options(Map("header" -> "true", "delimiter" -> ";",
"inferSchema" -> "true")).format("csv").load("C:/data/OpenData-gouv.fr/bal/adresses-
france.csv")
.withColumn("dept", regexp_replace(col("commune_insee"), "0*(.*)...", "$1").cast("int"))
.withColumn("code", col("commune_insee").cast("int"))

allAdressCsvDs.repartition(1)
 .sort("dept").sortWithinPartitions("dept", "code", "voie_nom")
 .write.format("parquet").option("parquet.block.size", 16*1024*1024)
 .saveAsTable("db1.address_sorted")

allAdressCsvDs.repartition(10, col("dept"))
 .sortWithinPartitions("dept", "code", "voie_nom")
 .write.format("parquet").option("parquet.block.size", 16*1024*1024)
 .saveAsTable("db1.address_sorted10")
```

Exercise 21: check.. getNumPartitions

Check that

a/ table db1.address_sorted is saved
on 1 parquet file,
and file is split into 55 partitions
giving a total of numPartition=55

b/ table db1.address_sorted10 is saved
on 10 parquet files
and files are split (differently..)
giving a total of numPartition=57

c/ is there a big difference between a/ and b/ ?
Which is « better » ?

Hint: use **dataset.toJavaRDD.getNumPartitions**

Exercise 22: Predicate-Push-Down

Execute several Queries with WHERE clauses « field=value »
and see efficiency of Skipped/Read bytes compared to total file size

select count(*) from db1.address_sorted where …
a/ where commune_nom='Nanterre'
b/ where code=92050
c/ where commune_nom='_UneCommuneQuiNExistePas'
d/ where code=0

Compare Bytes read for queries… Are they same?
Compare Plan « PushedFilters: [..] » / Compare jobs->stage tasks counts

Hint: use « **sc.setCallSite**(« DisplayName »); » to find queries more easily in Spark UI

Objectives of Hands-On

1/ Execution Plan, SQL Explain / dataset.explain

2/ Spark UI, DAG, Narrow/Wide Transformations

3/ RDD cache / checkpoint

4/ Partition Pruning / Columns Pruning / Predicate-Push-Down

5/ Parquet Optims (Sort, Stats, Dictionary, Bloom, BlockSize)

6/ perfs: SpillToDisk, JOINs, hint

Exercise 23: Detect SpillToDisk problems

When Spark has not enough RAM memory, it uses « swapping » to disk.
In Spark, this is called « SpillToDisk ».
This is a dreadfull signal, to ask for more memory (or better algorithm)

a/ execute a task taking to much memory
… example a « .repartition(2).sortWithinPartitions(« commune_insee »)

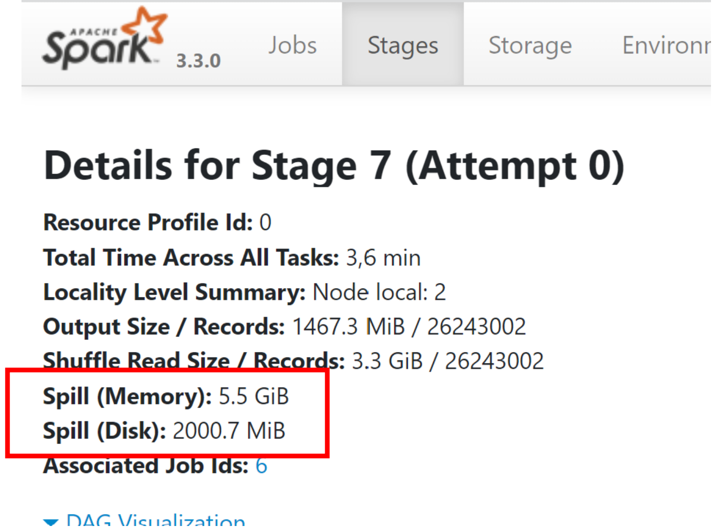
b/ detect the slowness and diagnostic it in SparkUI

c/ retry executing same request with more memory, and compare performances

Exercise 23: SpillToDisk

Example:

```
spark.sql("select * from db1.addr3")
  .repartition(2).sortWithinPartitions("commune_insee")
  .write.saveAsTable("db1.addr_repart2")
```



Find other (more detailed) infos in SparkUI

Exercise 24: Join

a/ Extract and save from table address a new table « city »
Containing « name, code, dept, average_address_longitute, average_address_lattitude »

b/ Join table « address » and « city »,
and compute for each address the offset longitude/latitude to the city average center

c/ study Execution plan
check that Spark is already efficiently « Broadcasting » the small 'city' table
=> no need to « SortMerge » the big 'addr' table

Exercise 25: HINT to force join strategy

compare with previous Exercise 24/, but with SQL hinting
SELECT /*+ MERGE(a) */ ...   FROM addr a ...

Cf https://spark.apache.org/docs/latest/sql-ref-syntax-qry-select-hints.html

Examples

```
-- Join Hints for broadcast join
SELECT /*+ BROADCAST(t1) */ * FROM t1 INNER JOIN t2 ON t1.key = t2.key;
SELECT /*+ BROADCASTJOIN (t1) */ * FROM t1 left JOIN t2 ON t1.key = t2.key;
SELECT /*+ MAPJOIN(t2) */ * FROM t1 right JOIN t2 ON t1.key = t2.key;

-- Join Hints for shuffle sort merge join
SELECT /*+ SHUFFLE_MERGE(t1) */ * FROM t1 INNER JOIN t2 ON t1.key = t2.key;
SELECT /*+ MERGEJOIN(t2) */ * FROM t1 INNER JOIN t2 ON t1.key = t2.key;
SELECT /*+ MERGE(t1) */ * FROM t1 INNER JOIN t2 ON t1.key = t2.key;

-- Join Hints for shuffle hash join
SELECT /*+ SHUFFLE_HASH(t1) */ * FROM t1 INNER JOIN t2 ON t1.key = t2.key;

-- Join Hints for shuffle-and-replicate nested loop join
SELECT /*+ SHUFFLE_REPLICATE_NL(t1) */ * FROM t1 INNER JOIN t2 ON t1.key = t2.key;
```

Optional Exercise 26: Default Shuffle = 200 ?!

Study number of partitions after a (« merge ») join (not a « broadcastjoin »)

a/ is it 200 ? Why

b/ How do you change default ?

c/ What should you always do before saving to File(s)?
(to avoid 200 small files)

Exercise 27 : MindMap

Draw a MindMap to summarize
what you did and learn from this Hands-On session

Your MindMap should
start with word « Spark – Optimizations & RDD DAGs» in the middle
Then draw star edges to other word chapters and sub-chapters

Objectives of Hands-On

- 1/ Execution Plan, SQL Explain / dataset.explain
- 2/ Spark UI, DAG, Narrow/Wide Transformations
- 3/ RDD cache / checkpoint
- 4/ Partition Pruning / Columns Pruning / Predicate-Push-Down
- 5/ Parquet Optims (Sort, Stats, Dictionary, Bloom, BlockSize)
- 6/ perfs: SpillToDisk, JOINs, hint

Questions ?

Take-Away

What You learned ?

Next Steps

Unfortunatly,
NO More Lessons
NO More Hands-On

But still, many more Spark concepts to learn :

- Spark Clustering
- Spark on Kubernetes, in Cloud
- Java binding, UDF, map
- Analytical Queries
- Machine Learning
- Spark Streaming
- …