Transformations

- The core data structures in Spark are **immutable**
- Spark applications process data using methods that describe what needs to be done
- These operations are called **transformations**

Transformations – Example

```
scala> val myRange = spark.range(1000).toDF("number")
myRange: org.apache.spark.sql.DataFrame = [number: bigint]
```

Transformations – Example

```
scala> val myRange = spark.range(1000).toDF("number")
myRange: org.apache.spark.sql.DataFrame = [number: bigint]
```

- Apply a transformation to find all the even numbers in the DF myRange

```
scala> val divisBy2 = myRange.where("number % 2 = 0")
divisBy2: org.apache.spark.sql.Dataset[org.apache.spark.sql.Row] = [number: bigint]
```

Transformations – Example

```
scala> val myRange = spark.range(1000).toDF("number")
myRange: org.apache.spark.sql.DataFrame = [number: bigint]
```

- Apply a transformation to find all the even numbers in the DF myRange

```
scala> val divisBy2 = myRange.where("number % 2 = 0")
divisBy2: org.apache.spark.sql.Dataset[org.apache.spark.sql.Row] = [number: bigint]
```

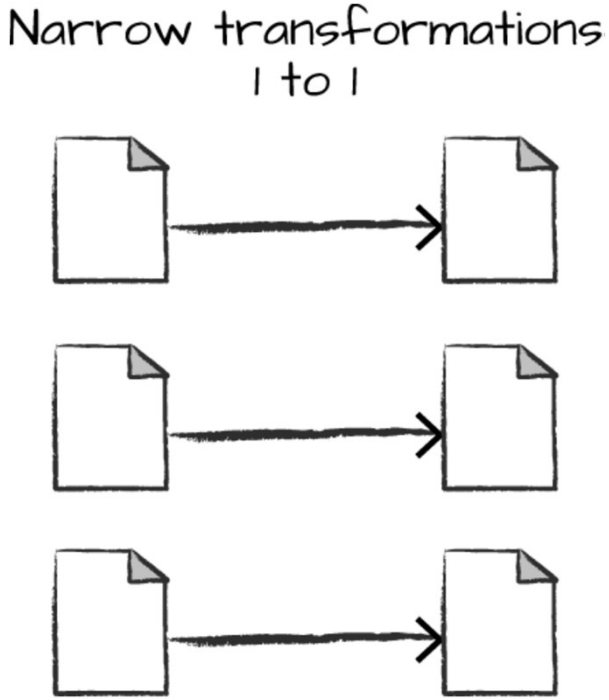
- No data will be output until we apply an action

Types of Transformations

- There are two types of transformations:
  1. narrow transformations
  2. wide transformations

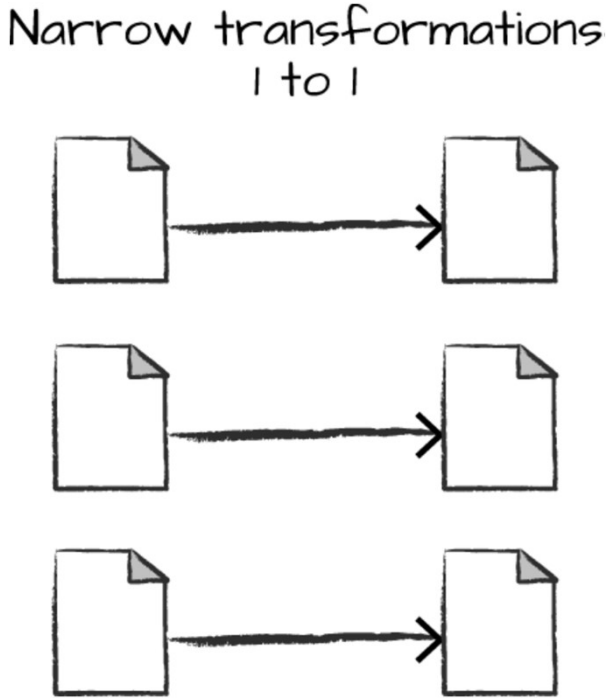
Narrow Transformations

- where each input partition contributes to only one output partition



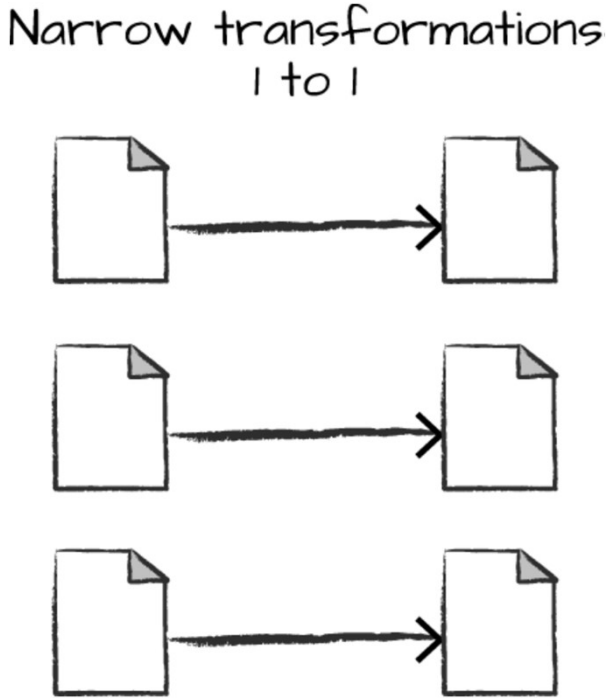
Narrow Transformations

- where each input partition contributes to only one output partition
- aka narrow dependencies



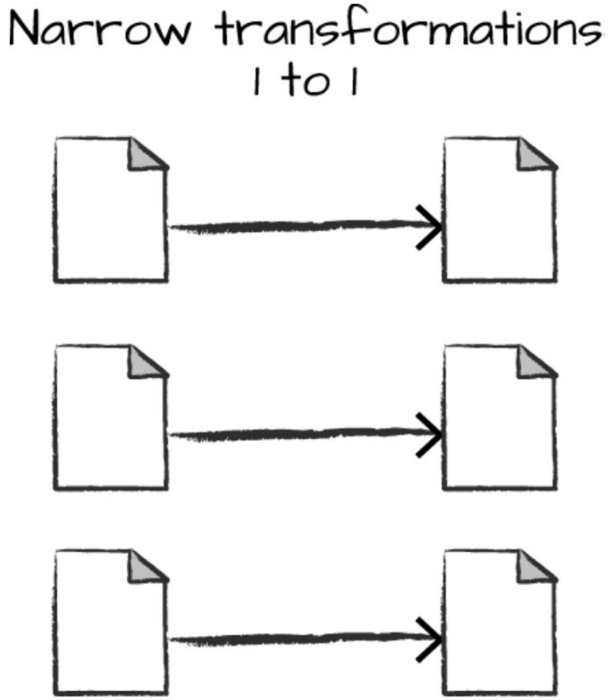
Narrow Transformations

- where each input partition contributes to only one output partition
- aka narrow dependencies
- examples: previous **where** statement, **filter**



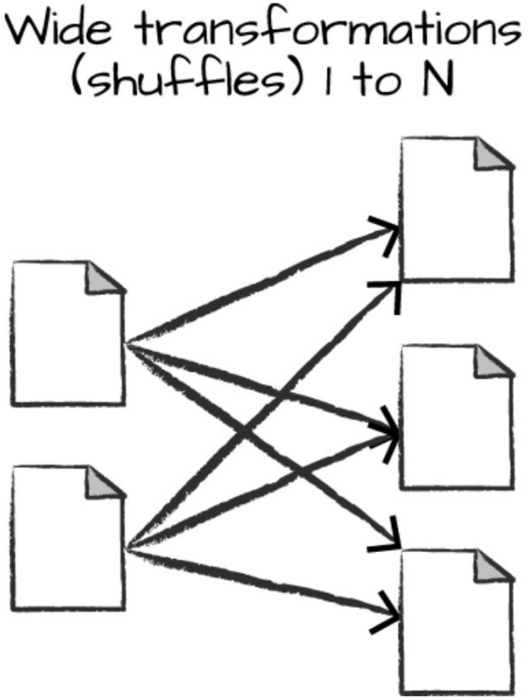
Narrow Transformations

- where each input partition contributes to only one output partition
- aka narrow dependencies
- examples: previous **where** statement, **filter**
- allow pipelining



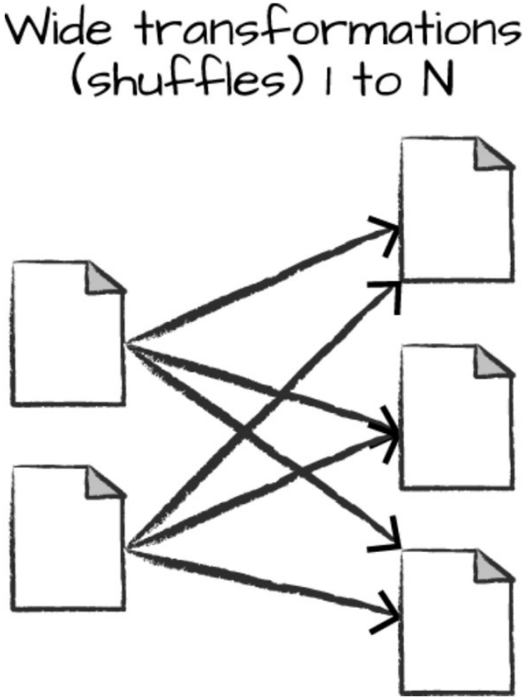
Wide Transformations

- where an input partition can contribute to more than one output partition



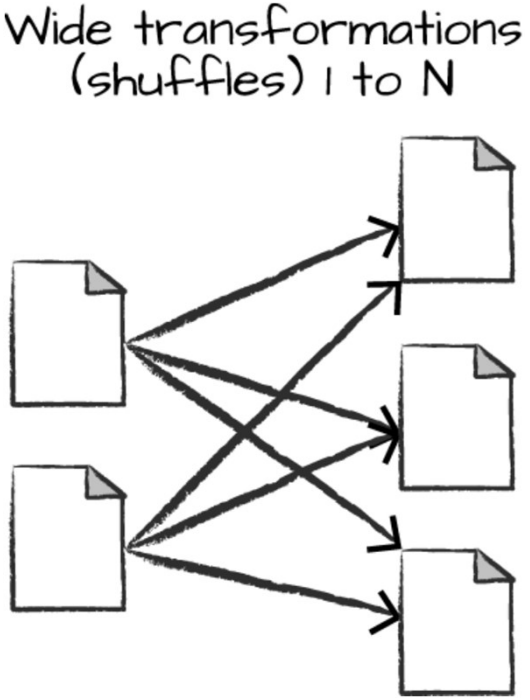
Wide Transformations

- where an input partition can contribute to more than one output partition
- aka wide dependencies



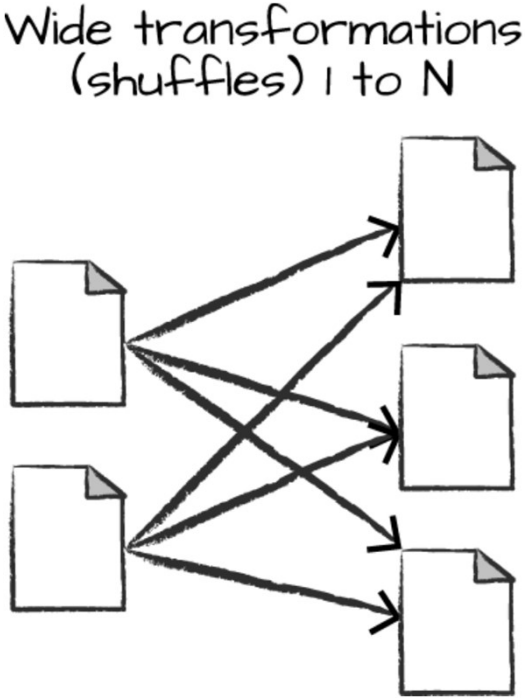
Wide Transformations

- where an input partition can contribute to more than one output partition
- aka wide dependencies
- examples: sort, groupByKey, join



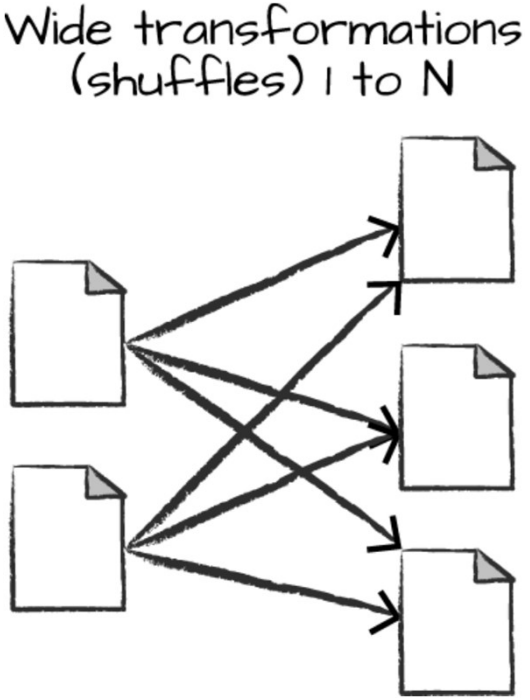
Wide Transformations

- where an input partition can contribute to more than one output partition
- aka wide dependencies
- examples: sort, groupByKey, join
- aka **shuffles**



Wide Transformations

- where an input partition can contribute to more than one output partition
- aka wide dependencies
- examples: sort, groupByKey, join
- aka **shuffles**
- a shuffle requires data to be written on **disk**



Lazy Evaluation

- Transformation operations are lazily evaluated
- Spark will delay their evaluations until the last minute

Lazy Evaluation

- Transformation operations are lazily evaluated
- Spark will delay their evaluations until the last minute
- Transformation specify a plan of operations to perform on the data

Lazy Evaluation

- Transformation operations are lazily evaluated
- Spark will delay their evaluations until the last minute
- Transformation specify a plan of operations to perform on the data
- Lazy evaluation allows Spark to optimize the code
  - perform a filter early or combine many transformations into a single operation

Actions

- An action is an operation that requires Spark to compute a result
- An action operation triggers the computation of code

Actions

- An action is an operation that requires Spark to compute a result
- An action operation triggers the computation of code
- Actions are computed **eagerly**

Actions

- An action is an operation that requires Spark to compute a result
- An action operation triggers the computation of code
- Actions are computed **eagerly**
- Example:

```
divisBy2.count()  // res3: Long = 500
```

Kinds of Actions

Kinds of Actions

- actions to view data in the console

Kinds of Actions

- actions to view data in the console
  - myRange.show()

Kinds of Actions

- actions to view data in the console
  - myRange.show()
- actions to collect data to native objects in the respective language

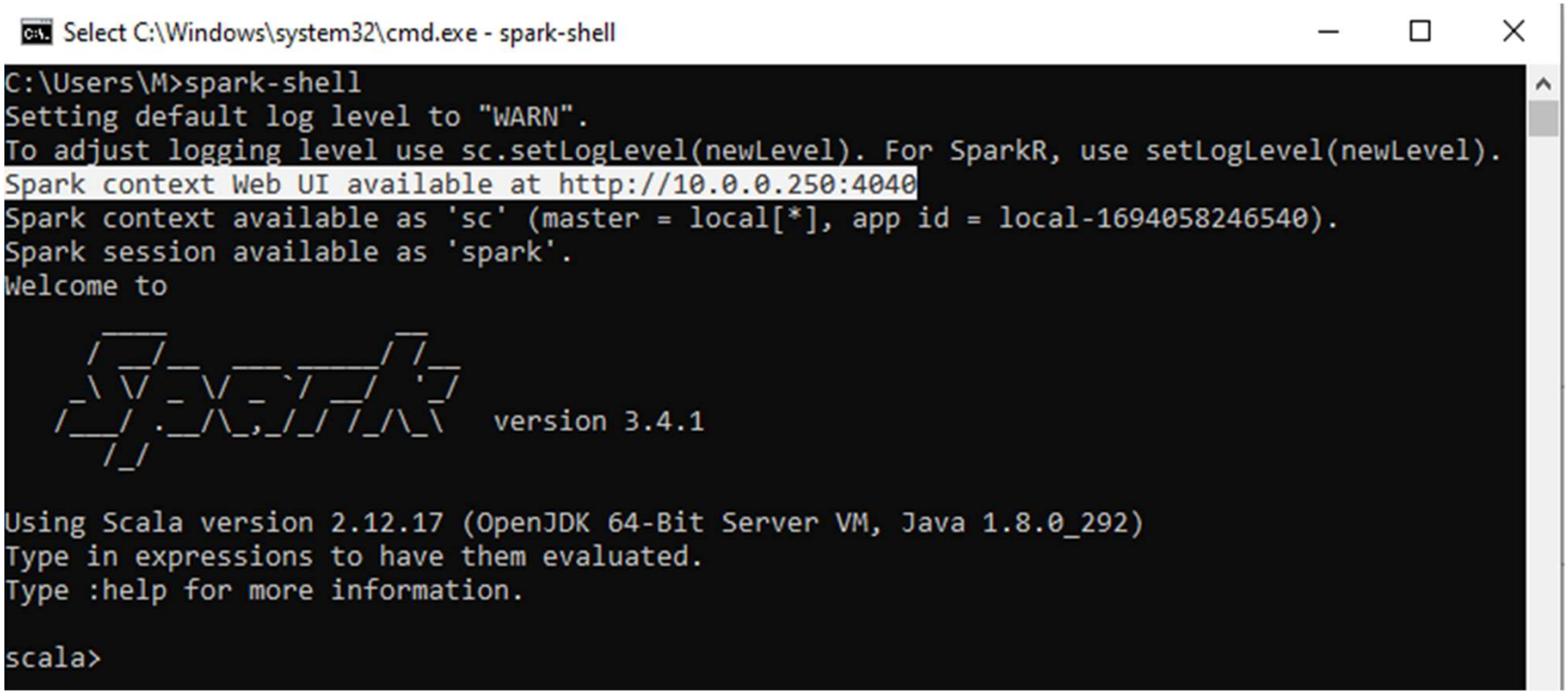
Kinds of Actions

- actions to view data in the console
  - myRange.show()
- actions to collect data to native objects in the respective language
  - myRange.collect()

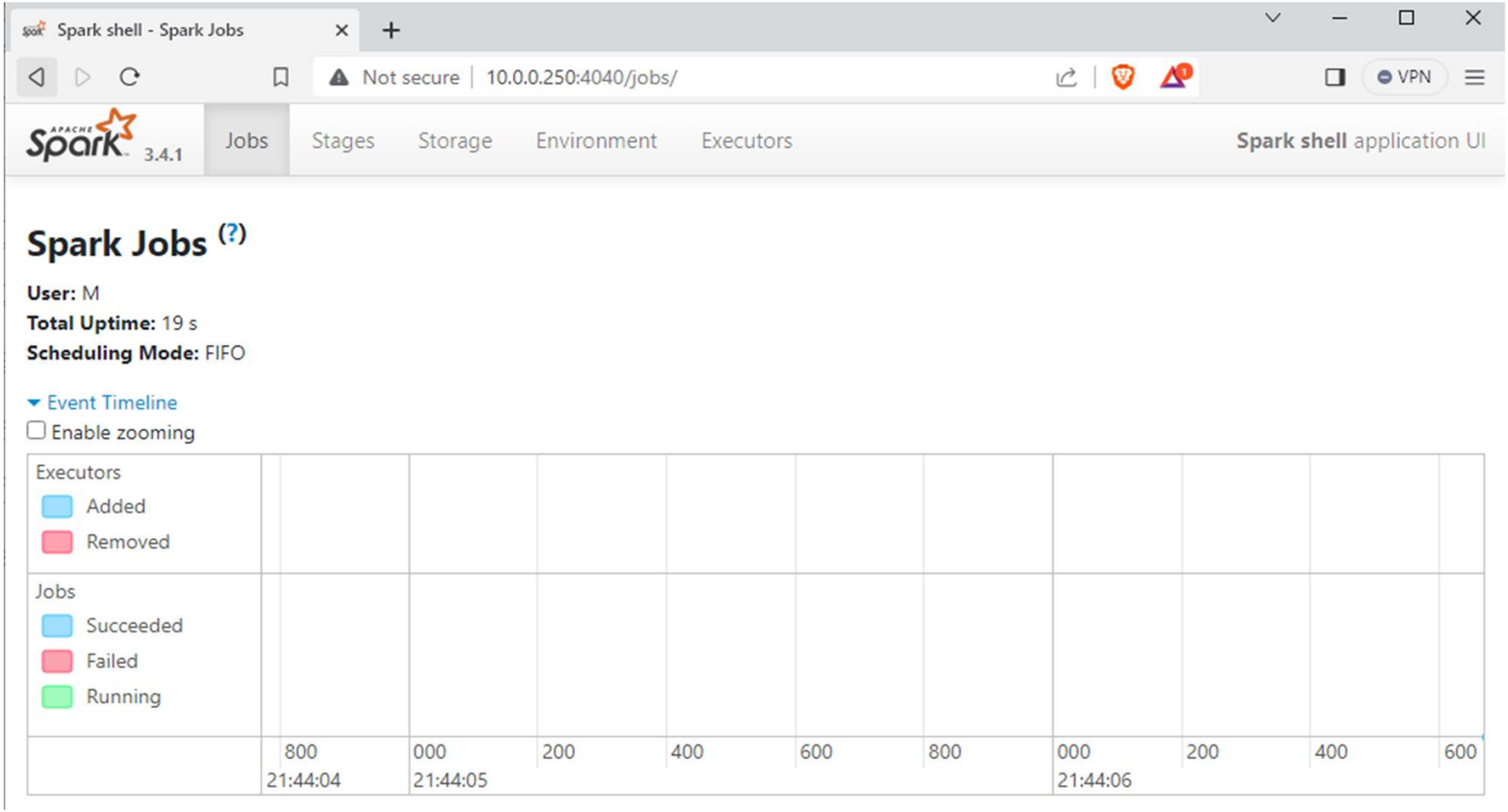
Kinds of Actions

- actions to view data in the console
  - myRange.show()
- actions to collect data to native objects in the respective language
  - myRange.collect()
- actions to write output to data sources

Spark UI


Select C:\Windows\system32\cmd.exe - spark-shell
C:\Users\M>spark-shell
Setting default log level to "WARN".
To adjust logging level use sc.setLogLevel(newLevel). For SparkR, use setLogLevel(newLevel).
Spark context Web UI available at http://10.0.0.250:4040
Spark context available as 'sc' (master = local[*], app id = local-1694058246540).
Spark session available as 'spark'.
Welcome to
version 3.4.1
Using Scala version 2.12.17 (OpenJDK 64-Bit Server VM, Java 1.8.0_292)
Type in expressions to have them evaluated.
Type :help for more information.
scala>

Spark UI


Spark shell - Spark Jobs
Not secure | 10.0.0.250:4040/jobs/
VPN
Spark 3.4.1
Jobs
Stages
Storage
Environment
Executors
Spark shell application UI
Spark Jobs (?)
User: M
Total Uptime: 19 s
Scheduling Mode: FIFO
Event Timeline
Enable zooming
Executors
Added
Removed
Jobs
Succeeded
Failed
Running
800
21:44:04
000
21:44:05
200
400
600
800
000
21:44:06
200
400
600

Spark Jobs

- A job is a **set** of computations (tasks) that Spark performs to return the results of an action to a driver program
- An action triggers a job
- A job is completed when a result is returned to a driver program
- An application can launch one or more jobs

Stage

- Spark breaks a job into a **DAG** of stages using shuffle boundaries
- Tasks that do not require a shuffle are grouped into the same stage.
- A task that requires its input data to be shuffled begins a new stage

DF’s explain Method

- prints the physical plan

DF's explain Method

- prints the physical plan
- lets us see the DF's **lineage**. i.e., how spark will execute the query to create the DF
- useful for debugging

```
val flightData2015 = spark.read.csv("someFile.csv")
```

DEST_COUNTRY_NAME	ORIGIN_COUNTRY_NAME	count
United States	Romania	1
United States	Ireland	264
United States	India	69

DF's explain Method

- prints the physical plan
- lets us see the DF's **lineage**. i.e., how spark will execute the query to create the DF
- useful for debugging

```
val flightData2015 = spark.read.csv("someFile.csv")
flightData2015.sort("count").explain()
```

DEST_COUNTRY_NAME	ORIGIN_COUNTRY_NAME	count
United States	Romania	1
United States	Ireland	264
United States	India	69

DF's explain Method

- prints the physical plan
- lets us see the DF's **lineage**. i.e., how spark will execute the query to create the DF
- useful for debugging

```
val flightData2015 = spark.read.csv("someFile.csv")
flightData2015.sort("count").explain()
== Physical Plan ==
*Sort [count#195 ASC NULLS FIRST], true, 0
+- Exchange rangepartitioning(count#195 ASC NULLS FIRST, 200)
   +- *FileScan csv [DEST_COUNTRY_NAME#193,ORIGIN_COUNTRY_NAME#194,count#195] ...
```

DEST_COUNTRY_NAME	ORIGIN_COUNTRY_NAME	count
United States	Romania	1
United States	Ireland	264
United States	India	69

Configuring Number of Partitions for a Shuffle

- By default, Spark uses 200 partitions for a shuffle

Configuring Number of Partitions for a Shuffle

- By default, Spark uses 200 partitions for a shuffle
- The following code lets us change that:

```
spark.conf.set("spark.sql.shuffle.partitions", "5")
```

Configuring Number of Partitions for a Shuffle

- By default, Spark uses 200 partitions for a shuffle
- The following code lets us change that:

```
spark.conf.set("spark.sql.shuffle.partitions", "5")
```

- Now, execute an explain plan

Configuring Number of Partitions for a Shuffle

- By default, Spark uses 200 partitions for a shuffle
- The following code lets us change that:

```
spark.conf.set("spark.sql.shuffle.partitions", "5")
```

- Now, execute an explain plan

```
flightData2015.sort("count").explain
*Sort [count#39 ASC NULLS FIRST], true, 0
+- Exchange rangepartitioning(count#39 ASC NULLS FIRST, 5)
   +- *(1) FileScan csv
[DEST_COUNTRY_NAME#37,ORIGIN_COUNTRY_NAME#38,count#39]
```

DataFrames and SQL

- You can express your business logic using SQL or DFs
- There is no performance difference
- Both “compile” to the same underlying plan specified in the DataFrame code
- Spark will run the same transformations either way

DataFrames and SQL (cont.)

- To write SQL, you need to first register the DF as a table / view

```
flightData2015.createOrReplaceTempView("flight_data_2015")
```

DataFrames and SQL (cont.)

- To write SQL, you need to first register the DF as a table / view

```
flightData2015.createOrReplaceTempView("flight_data_2015")
```

- Then, use the spark.sql function to write SQL code

```
val sqlWay = spark.sql("""
SELECT DEST_COUNTRY_NAME, count(1)
FROM flight_data_2015
GROUP BY DEST_COUNTRY_NAME
""")
```

DataFrames and SQL (cont.)

- To write SQL, you need to first register the DF as a table / view

```
flightData2015.createOrReplaceTempView("flight_data_2015")
```

- Then, use the spark.sql function to write SQL code

```
val sqlWay = spark.sql("""
SELECT DEST_COUNTRY_NAME, count(1)
FROM flight_data_2015
GROUP BY DEST_COUNTRY_NAME
""")
```

```
sqlWay:org.apache.spark.sql.DataFrame =
            [DEST_COUNTRY_NAME: string, count(1): long]
```

DataFrames and SQL (cont.)

```
val dataFrameWay = flightData2015
  .groupBy('DEST_COUNTRY_NAME)
  .count()
```

```
dataFrameWay:org.apache.spark.sql.DataFrame =
  [DEST_COUNTRY_NAME: string, count: long]
```

DataFrames and SQL (cont.)

```
val dataFrameWay = flightData2015
  .groupBy('DEST_COUNTRY_NAME)
  .count()
```

```
dataFrameWay:org.apache.spark.sql.DataFrame =
  [DEST_COUNTRY_NAME: string, count: long]
```

DataFrames and SQL (cont.)

```
sqlWay.explain
dataFrameWay.explain
```

```
== Physical Plan ==
*HashAggregate(keys=[DEST_COUNTRY_NAME#182],
functions=[count(1)])
+- Exchange hashpartitioning(DEST_COUNTRY_NAME#182, 5)
+- *HashAggregate(keys=[DEST_COUNTRY_NAME#182],
functions=[partial_count(1)])
+- *FileScan csv [DEST_COUNTRY_NAME#182] ...
== Physical Plan ==
*HashAggregate(keys=[DEST_COUNTRY_NAME#182],
functions=[count(1)])
+- Exchange hashpartitioning(DEST_COUNTRY_NAME#182, 5)
+- *HashAggregate(keys=[DEST_COUNTRY_NAME#182],
functions=[partial_count(1)])
+- *FileScan csv [DEST_COUNTRY_NAME#182] ...
```

Another Example

- Use the max function, to find the maximum number of flights to and from any given location, over all locations

Another Example

- Use the max function, to find the maximum number of flights to and from any given location, over all locations

```
spark.sql("SELECT max(count) from flight_data_2015").take(1)
```

Another Example

- Use the max function, to find the maximum number of flights to and from any given location, over all locations

```
spark.sql("SELECT max(count) from flight_data_2015").take(1)
```

```
Array([370002]
```

Another Example

- Use the max function, to find the maximum number of flights to and from any given location, over all locations

```
spark.sql("SELECT max(count) from flight_data_2015").take(1)
```

```
Array([370002]
```

```
import org.apache.spark.sql.functions.max

flightData2015.select(max("count")).take(1)
```

Another Example

- Use the max function, to find the maximum number of flights to and from any given location, over all locations

```
spark.sql("SELECT max(count) from flight_data_2015").take(1)
```

```
Array([370002]
```

```
import org.apache.spark.sql.functions.max

flightData2015.select(max("count")).take(1)
```

```
import org.apache.spark.sql.functions.max
Array([370002])
```

Example – Using Multiple Transformations

- Find the top five **destination** countries in the data

```
val maxSql = spark.sql("""
SELECT DEST_COUNTRY_NAME, sum(count) as destination_total
FROM flight_data_2015
GROUP BY DEST_COUNTRY_NAME
ORDER BY sum(count) DESC
LIMIT 5
""")

maxSql.show()
```

Example – Using Multiple Transformations

- Find the top five **destination** countries in the data

```
val maxSql = spark.sql("""
SELECT DEST_COUNTRY_NAME, sum(count) as destination_total
FROM flight_data_2015
GROUP BY DEST_COUNTRY_NAME
ORDER BY sum(count) DESC
LIMIT 5
""")

maxSql.show()
```

```
maxSql:org.apache.spark.sql.DataFrame =
        [DEST_COUNTRY_NAME: string, destination_total: long]

+-----------------------------+-----------------+
|DEST_COUNTRY_NAME|destination_total |
+-----------------------------+-----------------+
| United States               |           411352|
| Canada                      |             8399|
| Mexico                      |             7140|
| United Kingdom              |             2025|
| Japan                       |             1548|
+-----------------------------+-----------------+
```

Using Multiple Transformations -- DataFrame Syntax

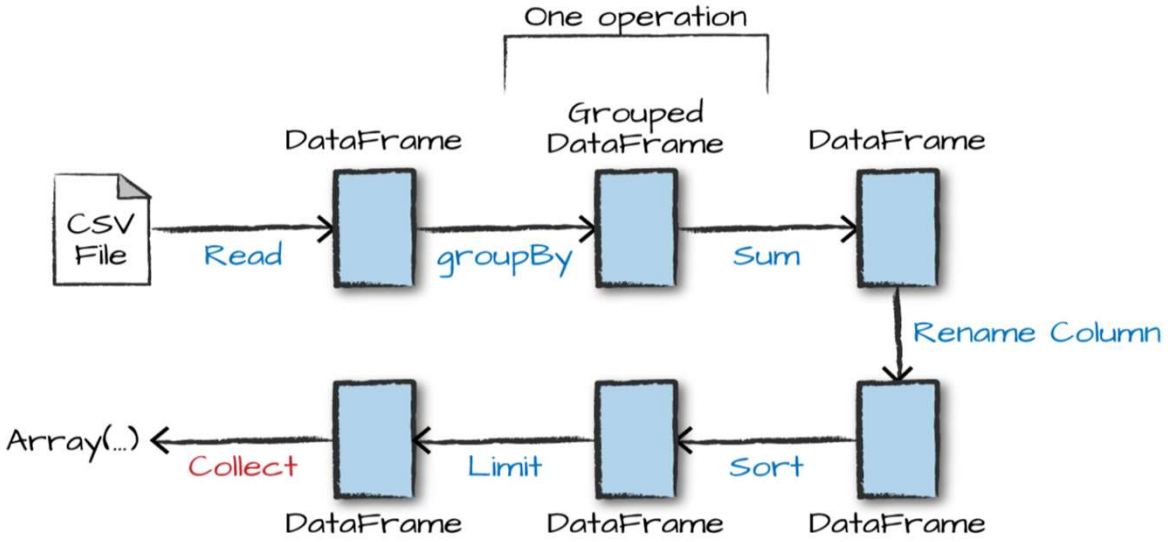
```
import org.apache.spark.sql.functions.desc

flightData2015
  .groupBy("DEST_COUNTRY_NAME")
  .sum("count")
  .withColumnRenamed("sum(count)","destination_total")
  .sort(desc("destination_total"))
  .limit(5)
  .show()
```

Using Multiple Transformations -- DataFrame Syntax

```
import org.apache.spark.sql.functions.desc

flightData2015
  .groupBy("DEST_COUNTRY_NAME")
  .sum("count")
  .withColumnRenamed("sum(count)","destination_total")
  .sort(desc("destination_total"))
  .limit(5)
  .show()
```



The entire DataFrame transformation flow