PySpark

- Spark is a unified engine and a set of libraries for parallell data processing on computer clusters. Spark is written in Scala and runs on Java Virtual Machine (JVM)
- PySpark is the python API for Spark, an distributed computing system used for Big Data processing and analytics. Spark offers a UI in http://localhost:4040
  - Cluster, or group, of computer, pools the resources of many machines together

1. Starting a spark session:

```
from pyspark.sql import SparkSession
spark = SparkSession.builder.appName("Insert a name").getOrCreate().master("local[*]")
```

Parallelism using all local cores

2. Loading data

```
df = spark.read.(parquet/csv)("file.csv", header=True, inferSchema=True)
df.show()
```

Spark only executes when actions are called
- .show()
- .collect()
- .count()

Operation	Pandas	PySpark
Select columns	`df[["column1", "column2"]]`	`df.select("column1", "column2")`
Filter rows	`df[df["column1"]>10]`	`df.filter(df.column1>10)`
Multiple conditions	`df[(df["col1"]>10)&(df["col2"]<10)]`	`df.filter((df.col1>10)&(df.col2<10))`
Apply a function to column	`df["col1"].apply(lambda x: x+1)`	`df.withColumn("col1", df.col1+1)`
Add a new column	`df["new"]=df["old"]+1`	`df.withColumn("new", df.old+1)`
Drop column(s)	`df.drop(["col1"], axis=1)`	`df.drop("col1")`
Rename column(s)	`df.rename(columns={"a":"b"})`	`df.withColumnRenamed("a","b")`
Replace values	`df.replace(0, np.nan)`	`df.replace(0, None)`
Drop missing values	`df.dropna()`	`df.na.drop()`
Fill missing values	`df.fillna(0)`	`df.na.fill(0)`

Operation	Pandas	PySpark
Group by & aggregate	`df.groupby("col1").sum()`	`df.groupBy("col1").sum()`
Join	`pd.merge(df1, df2, on='id')`	`df1.join(df2, on='id', how='inner')`
Sort	`df.sort_values("col1")`	`df.orderBy("col1")`
Distinct values	`df["col1"].unique()`	`df.select("col1").distinct()`
Count rows	`len(df)` or `df.shape[0]`	`df.count()`
Head / Preview	`df.head()`	`df.show(5)`
Schema info	`df.info()`	`df.printSchema()`

- How Spark works?

Spark consists of
- Driver process: Runs main(), assign the work for executors
- Executor process: Carry out the work assigned by drivers

Cluster manager: keep track of the resources available

Executor process always runs in Spark code

Driver process can run in different languages through Spark's language APIs

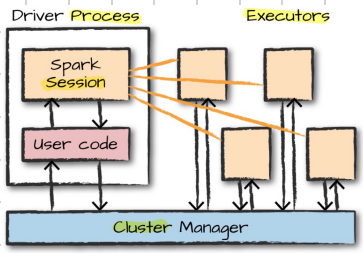


Figure 2-1. The architecture of a Spark Application

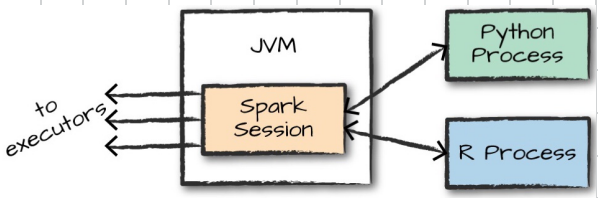


Figure 2-2. The relationship between the SparkSession and Spark's Language API

- Spark session translate my python code and then can run on the JVM executor
- To allow each executor to perform in parallel, Spark breaks up the data into chunks called partitions
- A partition is a set of rows that sit on one physical machine in my cluster

- Transformations: Operations that define a dataset but don't execute anything yet. Spark build a plan with them and wait for an action
- Actions: kick off the actual computation of my transformations
  - Transformation: .select(), .withColumn(), .filter()
  - Actions: .show(), .collect(), .count()

Types of transformations

- Narrow transformations: Each input partition will contribute to only one output partition
- Wide transformations: Each input partition contribute to many output partitions

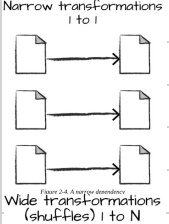


Figure 2-4. A narrow dependency

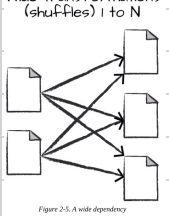


Figure 2-5. A wide dependency

Type of actions

- Action to view data
- Action to collect data
- Action to write to output data sources

.explain(): Shows the execution plan for a dataframe

```
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] ...
```

Top: Ending result
Bottom: The source data

Dataframe and SQL

- With Spark SQL, we can register any dataframe as a table and query it using SQL. There is not performance difference
- We must use spark.sql function.

1. Read

```
flightData = spark.read.csv("data", inferSchema="True"...)
```

2. Queries

```
data_sql = spark.sql("""SELECT DEST_COUNTRY_NAME, count(1)
FROM flightData
GROUP BY DEST_COUNTRY_NAME
""")

data = flightData.groupBy("DEST_COUNTRY_NAME").count()
```

3. Comparison

```
data_sql.explain()
data.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] ...
```

Same physical plan, both options are equal

```
# in Python
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()
```

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

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

Original column — Final column

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

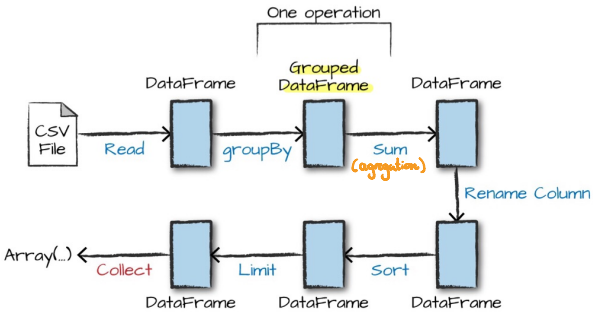


Figure 2-10. The entire DataFrame transformation flow

Arrows go in one direction → No loops

- This is a directed acyclic graph (DAG)
- Grouped dataframe: Group by group rows with the same value in one specific column (KEY) but we need to apply aggregation to specific what should we do with the rest of columns

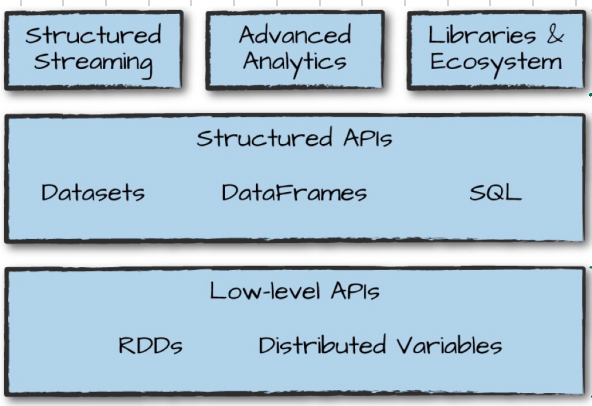
Vendor_id	fare_amount
1	15.00
2	5.0
1	10.00

GroupBy(Vendor_id)

Vendor_id	Fare amount
2	5.0
1	15 and 10

What should we do? Aggregation (sum)

- Sparks toolset



Libraries: Add functionalities

Treat data as structured table
- High-level (what I want but not how to do it)
- Automatically optimized
- Schema-aware (knows column names and types)

Datasets are replaced by dataframes in python

Gives fine-grained control and work on raw as objects
- Low-level (how to do it)
- Manual optimization
- No schema

Figure 3-1. Spark's toolset

spark-submit: This command gives the option to run my code on a Spark cluster
- Local
- A standalone cluster
- YARN, Mesos or Kubernetes

```
spark-submit \
  --master local[*] \
  --name "NYC Taxi Project" \
  my_script.py
```

local[*]: Run using all CPU cores, without [*], only runs 1 CPU core
"NYC Taxi Project": Name of my Spark app (shows in Spark UI)
my_script.py: Name of my pyspark script

MLlib: Machine learning library for spark, like any ML algorithm, these algorithms require the data is expressed in numerical value

```
# in Python
from pyspark.sql.functions import date_format, col
preppedDataFrame = staticDataFrame\
  .na.fill(0)\
  .withColumn("day_of_week", date_format(col("InvoiceDate"), "EEEE"))\
  .coalesce(5)
```

new column
Old column

- .na.fill(0): Fill all nan or missing values with 0
- .withColumn(...): Creates new column using "InvoiceDate" and use transformation date
- .coalesce(5): Reduce the number of partition to 5

Is 2024-08-01 is monday return Monday

Spark divides large dataset into partitions to process them in parallel

- .coalesce(): Only reduces the number of partitions
- .Parallelize(): Increase or reduce the number of partitions

Split the dataset:

```
# in Python
trainDataFrame = preppedDataFrame\
  .where("InvoiceDate < '2011-07-01'")
testDataFrame = preppedDataFrame\
  .where("InvoiceDate >= '2011-07-01'")
```

- Where and filter are the same functions in Spark
- RandomSplit([0.X, 0.1-X]) is also valid (Train, Test)

Transform the data

```
# in Python
from pyspark.ml.feature import StringIndexer
indexer = StringIndexer()\
  .setInputCol("day_of_week")\
  .setOutputCol("day_of_week_index")
```

- StringIndexer: Transform categorical string columns to numerical columns sorting labels by frequency. If Tuesday and then Monday are the most common days: Tuesday: 0.0 Monday 1.0

indexer_df = indexer.fit(df).transform(df) (or use a pipeline)

One hot encoder: StringIndexer is not one hot encoder

```
# in Python
from pyspark.ml.feature import OneHotEncoder
encoder = OneHotEncoder()\
  .setInputCol("day_of_week_index")\
  .setOutputCol("day_of_week_encoded")
```

Tuesday 0.0 Monday 1.0 —one hot→ Tuesday [1.0, 0.0] Monday [0.0, 1.0]

Vector Assembler: All machine learning algorithms in Spark take as a input a Vector type

```
# in Python
from pyspark.ml.feature import VectorAssembler

vectorAssembler = VectorAssembler()\
  .setInputCols(["UnitPrice", "Quantity", "day_of_week_encoded"])\
  .setOutputCol("features")
```

- Takes these three columns as a input and combine them to create a vector column

UnitPrice	Quantity	Day_of_week_encoded	Features
1.0	0.0	0.5	[1.0,0.0,0.5]

Pipeline: is a way to chain together multiple data processing stages, reusable workflow

```
# in Python
from pyspark.ml import Pipeline

transformationPipeline = Pipeline()\
  .setStages([indexer, encoder, vectorAssembler])
```

(indexer: String_indexer, encoder: One-hot encoder)

Components of a Spark pipeline:
- Transformers: Applies transformations to the data (indexer, encoder and vector assembler)
- Estimators: Machine learning model that learn from data