

# Spark Physical and Logical Plan Analysis

## using spark (Scala & Python)

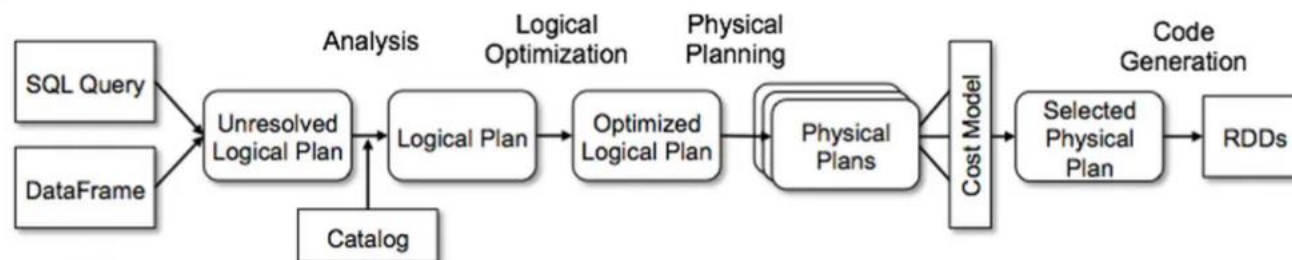


# What is Physical and Logical Plan?

**Spark Logical and physical plan** provides a visual representation of the data's journey, including all steps from origin to destination, with detailed information about where the data goes, who owns the data, and how the data is processed and stored at each step. Spark-Lineage extracts all necessary metadata from every Spark-ETL job. It is majorly used by developers for debugging purpose.

Here's how it works:

- The code written is first noted as an unresolved logical plan, if it is valid then Spark converts this into a logical plan
- The logical plan is passed through the Catalyst Optimizer to apply optimized rules.
- The Optimized Logical Plan is then converted into a Physical Plan
- The Physical Plan is executed by the Spark executors.



# Understanding with examples

We have used a financial dataset from Kaggle to analyse financial instrument monthly insights.

**URL:** <https://www.kaggle.com/datasets/hanseopark/sp-500-stocks-value-with-financial-statement>

**Case study:** we have written PySpark/Scala spark code for checking the monthly highs and lows for individual financial instrument and we will see how spark creates the physical plan and logical plan while executing the snippet of code.

**Note:** We will use spark explain function to understand the physical and logical plan.


`DataFrame.explain(extended=None, mode=None)[source]`

Prints the (logical and physical) plans to the console for debugging purpose.  
specifies the expected output format of plans:

- `explain(mode="simple")` – will display the physical plan
- `explain(mode="extended")` – will display physical and logical plans (like “extended” option)
- `explain(mode="codegen")` – will display the java code planned to be executed
- `explain(mode="cost")` – will display the optimized logical plan and related statistics (if they exist)
- `explain(mode="formatted")` – will display a split output composed of a nice physical plan outline and a section with each node details
- `explain(extended=False)` which projects only the physical plan
- `explain(extended=True)` which projects all the plans (logical and physical)

# PySpark example

We have used aggregation function to achieve the above problem statement.


 jupyter PysparkCodeBase Last Checkpoint: 4 hours ago (autosaved)












Logout

File Edit View Insert Cell Kernel Widgets Help

Trusted

Python 3 (ipykernel) 

         Run    Code 

```
In [27]: ▶ from pyspark.sql import SparkSession
from pyspark.sql.functions import min, max, col

#Spark session creation
spark = SparkSession.builder \
    .master("local") \
    .appName("PysparkCode") \
    .getOrCreate()

#Advice: I am using InferSchema just for test purpose, but I won't suggest to use it.
priceDf = spark.read.option('header', 'true').option('InferSchema', 'true').csv('FS_sp500_Value.csv')

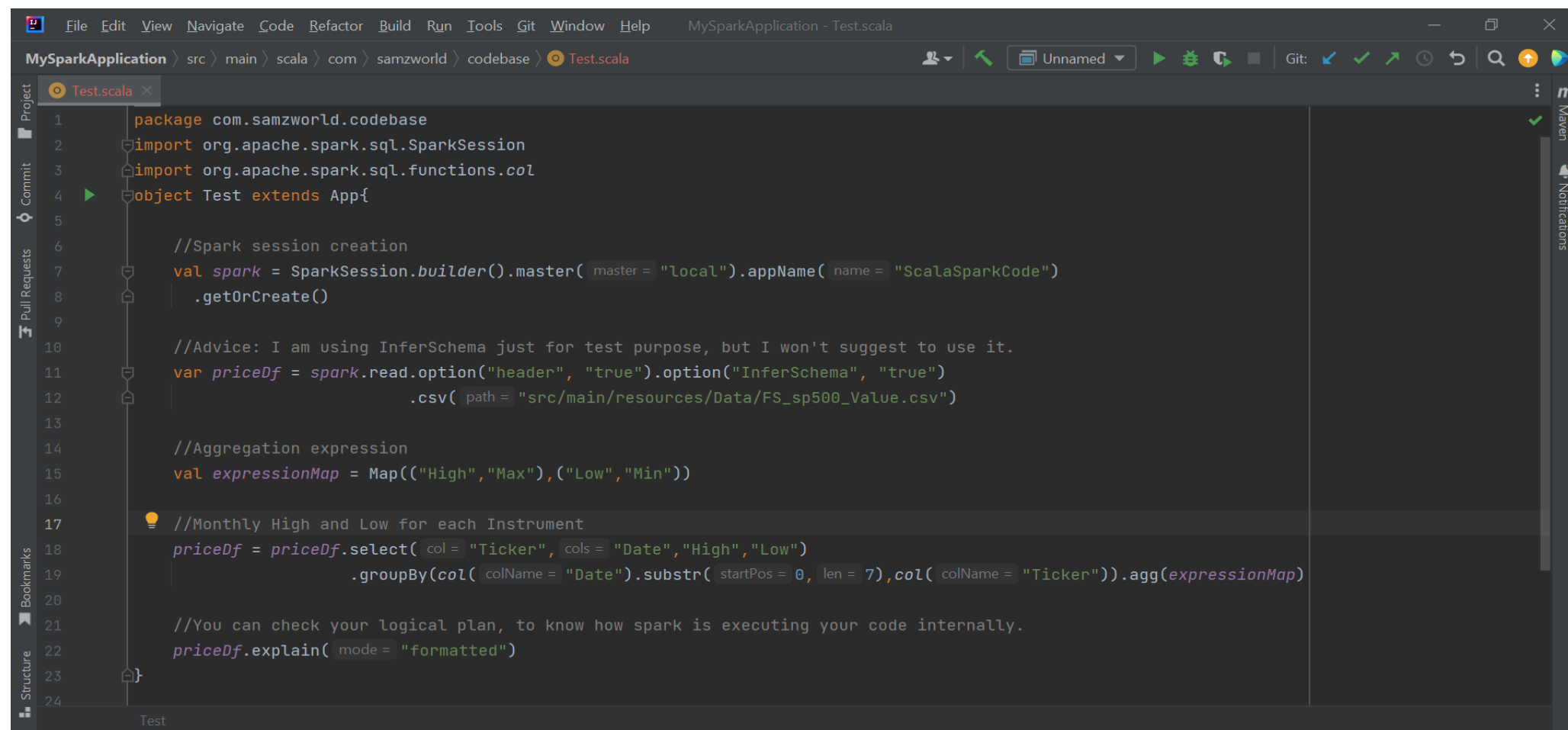
#Aggregation expression
expressionMap = {'High' : 'Max', 'Low' : 'Min'}

#Monthly High and Low for each Instrument
priceDf = priceDf.select("Ticker", "Date", "High", "Low").groupBy(col("Date").substr(0,7), col("Ticker")).agg(expressionMap)

#You can check your logical plan, to know how spark is executing your code internally.
priceDf.explain('extended')
```

# Scala Spark example

Similarly, we have used the below Scala Spark code for solving the problem statement.



```
1 package com.samzworld.codebase
2 import org.apache.spark.sql.{SparkSession, SparkContext}
3 import org.apache.spark.sql.functions.col
4 object Test extends App {
5
6     //Spark session creation
7     val spark = SparkSession.builder().master("local").appName("ScalaSparkCode")
8     .getOrCreate()
9
10    //Advice: I am using InferSchema just for test purpose, but I won't suggest to use it.
11    var priceDf = spark.read.option("header", "true").option("InferSchema", "true")
12    .csv(path = "src/main/resources/Data/FS_sp500_Value.csv")
13
14    //Aggregation expression
15    val expressionMap = Map(("High", "Max"), ("Low", "Min"))
16
17    //Monthly High and Low for each Instrument
18    priceDf = priceDf.select(col = "Ticker", cols = "Date", "High", "Low")
19    .groupBy(col(colName = "Date").substr(startPos = 0, len = 7), col(colName = "Ticker")).agg(expressionMap)
20
21    //You can check your logical plan, to know how spark is executing your code internally.
22    priceDf.explain(mode = "formatted")
23 }
24
```

# Physical Plan explained

If you see the explain("formatted") provided the whole data journey in a readable format, starting from reading csv, followed by arranging the column, followed by partial data aggregation, followed by data shuffling, followed by performing the final aggregation and then final results.

```
== Physical Plan ==
AdaptiveSparkPlan (6)
+- HashAggregate (5)
   +- Exchange (4)
      +- HashAggregate (3)
         +- Project (2)
            +- Scan csv (1)

(1) Scan csv
Output [4]: [Ticker#933, Date#934, High#935, Low#936]
Batched: false
Location: InMemoryFileIndex [file:/C:/Users/Shubham/FS_sp500_Value.csv]
ReadSchema: struct<Ticker:string,Date:timestamp,High:double,Low:double>

(2) Project
Output [4]: [Ticker#933, High#935, Low#936, substring(cast(Date#934 as string), 0, 7) AS _groupingexpression#968]
Input [4]: [Ticker#933, Date#934, High#935, Low#936]

(3) HashAggregate
Input [4]: [Ticker#933, High#935, Low#936, _groupingexpression#968]
Keys [2]: [_groupingexpression#968, Ticker#933]
Functions [2]: [partial_max(High#935), partial_min(Low#936)]
Aggregate Attributes [2]: [max#969, min#970]
Results [4]: [_groupingexpression#968, Ticker#933, max#971, min#972]

(4) Exchange
Input [4]: [_groupingexpression#968, Ticker#933, max#971, min#972]
Arguments: hashpartitioning(_groupingexpression#968, Ticker#933, 200), ENSURE_REQUIREMENTS, [id=#494]

(5) HashAggregate
Input [4]: [_groupingexpression#968, Ticker#933, max#971, min#972]
Keys [2]: [_groupingexpression#968, Ticker#933]
Functions [2]: [max(High#935), min(Low#936)]
Aggregate Attributes [2]: [max(High#935)#962, min(Low#936)#963]
Results [4]: [_groupingexpression#968 AS substring(Date, 0, 7)#959, Ticker#933, max(High#935)#962 AS max(High)#960, min(Low#936)#963 AS min(Low)#961]

(6) AdaptiveSparkPlan
Output [4]: [substring(Date, 0, 7)#959, Ticker#933, max(High)#960, min(Low)#961]
Arguments: isFinalPlan=false
```

# How I can see the logical plan along with physical plan?

Well, for that we need to change the explain argument to `priceDf.explain('extended')`. You can see the whole journey of logical plan to physical plan. Similarly you can try each parameters and learn about spark internal working.

```
== Parsed Logical Plan ==
'Aggregate [substring('Date, 0, 7), 'Ticker], [substring('Date, 0, 7) AS substring(Date, 0, 7)#1929, 'Ticker, 'max(High#1905) AS max(High)#1930, 'min(Low#1906) AS min(Low)#1931]
+- Project [Ticker#1903, Date#1904, High#1905, Low#1906]
   +- Relation [_c0#1902,Ticker#1903,Date#1904,High#1905,Low#1906,Open#1907,Close#1908,Volume#1909,Adj Close#1910] csv

== Analyzed Logical Plan ==
substring(Date, 0, 7): string, Ticker: string, max(High): double, min(Low): double
Aggregate [substring(cast(Date#1904 as string), 0, 7), Ticker#1903], [substring(cast(Date#1904 as string), 0, 7) AS substring(Date, 0, 7)#1929, Ticker#1903, max(High#1905) AS max(High)#1930, min(Low#1906) AS min(Low)#1931]
+- Project [Ticker#1903, Date#1904, High#1905, Low#1906]
   +- Relation [_c0#1902,Ticker#1903,Date#1904,High#1905,Low#1906,Open#1907,Close#1908,Volume#1909,Adj Close#1910] csv

== Optimized Logical Plan ==
Aggregate [_groupingexpression#1938, Ticker#1903], [_groupingexpression#1938 AS substring(Date, 0, 7)#1929, Ticker#1903, max(High#1905) AS max(High)#1930, min(Low#1906) AS min(Low)#1931]
+- Project [Ticker#1903, High#1905, Low#1906, substring(cast(Date#1904 as string), 0, 7) AS _groupingexpression#1938]
   +- Relation [_c0#1902,Ticker#1903,Date#1904,High#1905,Low#1906,Open#1907,Close#1908,Volume#1909,Adj Close#1910] csv

== Physical Plan ==
AdaptiveSparkPlan isFinalPlan=false
+- HashAggregate(keys=[_groupingexpression#1938, Ticker#1903], functions=[max(High#1905), min(Low#1906)], output=[substring(Date, 0, 7)#1929, Ticker#1903, max(High)#1930, min(Low)#1931])
   +- Exchange hashpartitioning(_groupingexpression#1938, Ticker#1903, 200), ENSURE_REQUIREMENTS, [id=#1005]
      +- HashAggregate(keys=[_groupingexpression#1938, Ticker#1903], functions=[partial_max(High#1905), partial_min(Low#1906)], output=[_groupingexpression#1938, Ticker#1903, max#1941, min#1942])
         +- Project [Ticker#1903, High#1905, Low#1906, substring(cast(Date#1904 as string), 0, 7) AS _groupingexpression#1938]
            +- FileScan csv [Ticker#1903,Date#1904,High#1905,Low#1906] Batched: false, DataFilters: [], Format: CSV, Location: InMemoryFileIndex(1 paths)[file:/C:/Users/Shubham/FS_sp500_Value.csv], PartitionFilters: [], PushedFilters: [], ReadSchema: struct<Ticker:string,Date:timestamp,High:double,Low:double>
```