Spark on YARN Architecture

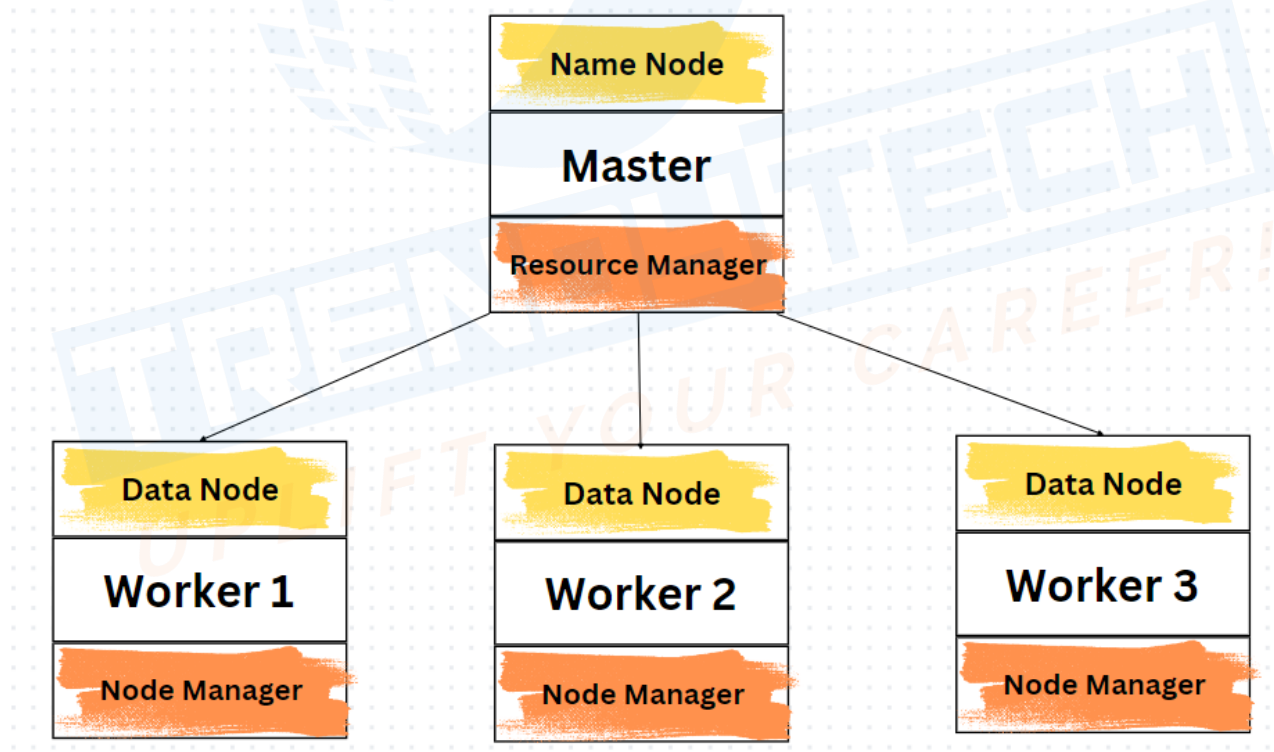
Hadoop Core Components :

HDFS	MAPREDUCE	YARN
Storage	Processing	Resource Manager

YARN Architecture

YARN consists of two major components

1. Resource Manager (Master)
2. Node Manage (Worker / Slave)



Name Node & Data Node related to HDFS (STORAGE)

Resource Manager & Node Manager related to YARN

Processes involved in invoking a Hadoop Job from Client machine

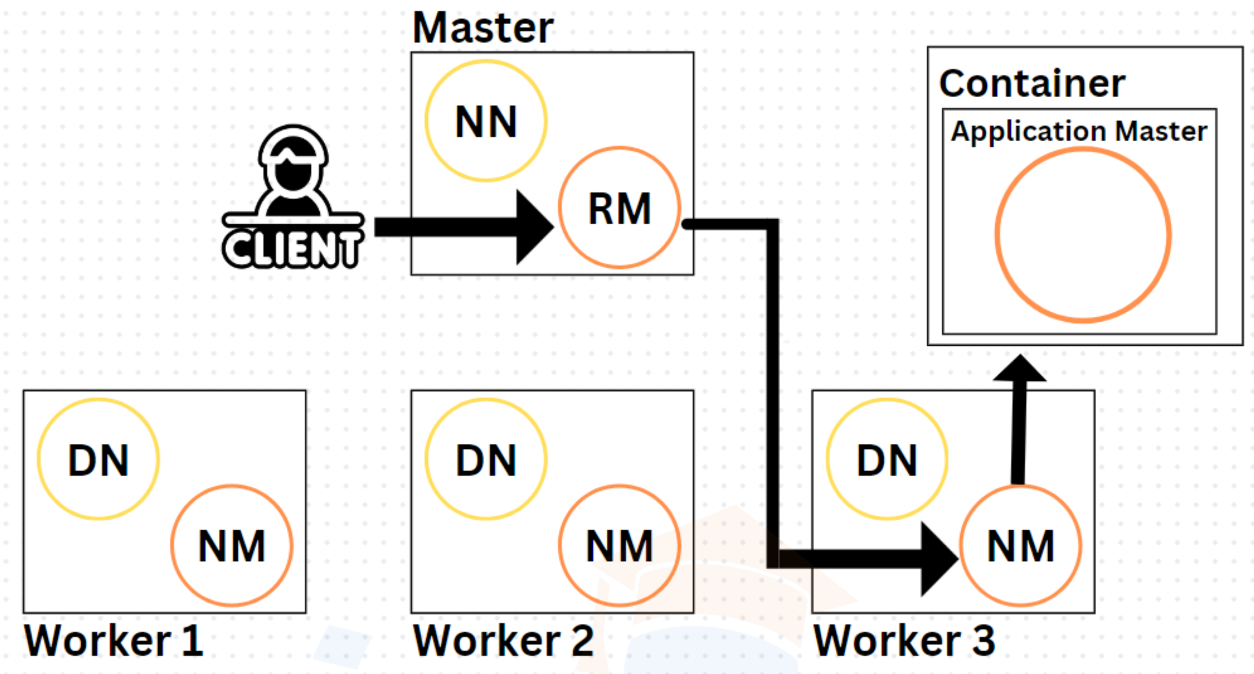
hadoop jar <Jar-Name>

On executing the above command on the client machine, following things happen :

1. **Request goes to Resource Manager**
2. **Resource Manager creates a Container in one of the Node Managers** (by coordinating with the Node Manager)
3. **An Application Master Service will be started inside this Container.** (Application Master is a local Manager that manages the application)

Note : Application Master is responsible to negotiate for the required resources from the Resource Manager. It will interact with the Name Node to know where the Data is located and accordingly request for resources on specific nodes as it works on the principle of Data Locality.

Every Application has an Application Master i.e., If there are 20 Applications, then there would be 20 Application Masters running.



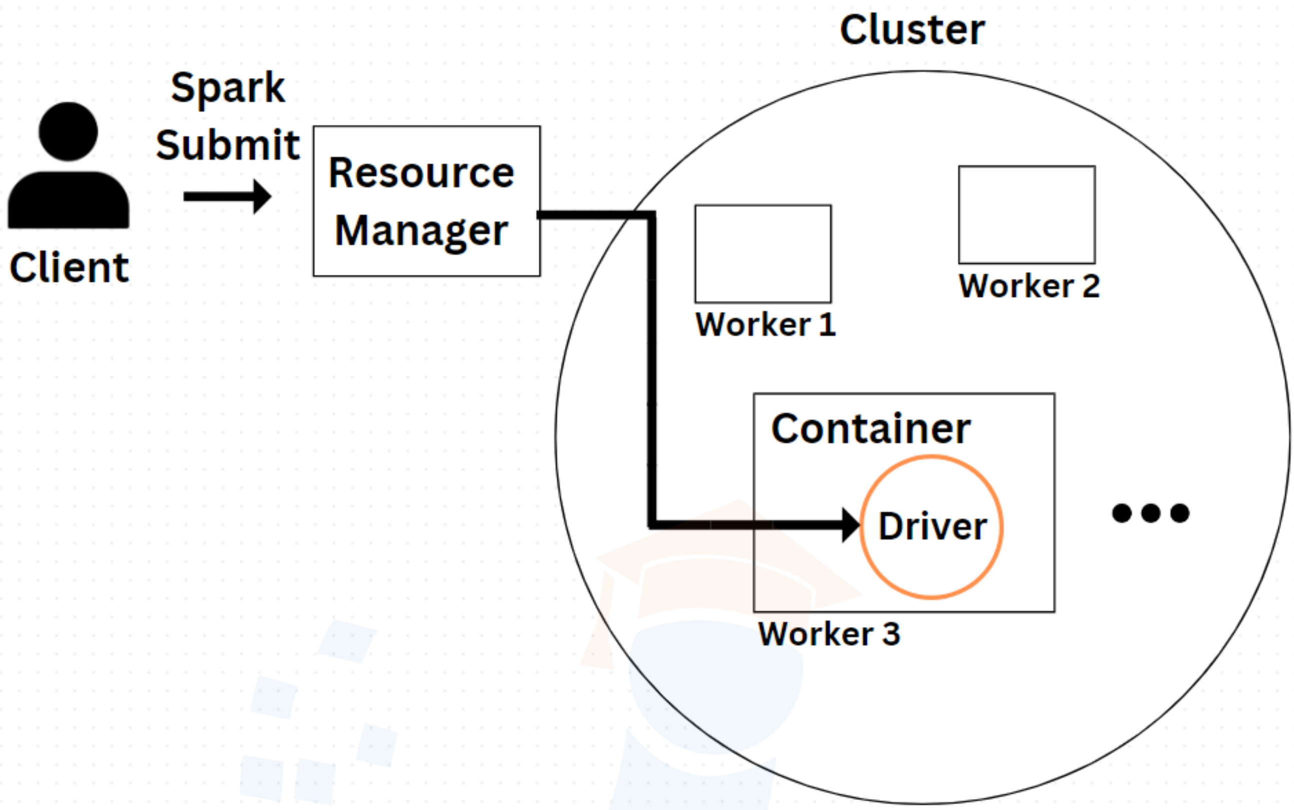
Uber Mode

Is a mode in which the application will be executed in the container where the application master is running. This mode is for scenarios where the job is small enough to be executed on a single container.

Two ways of running Spark Code-

1. **Interactive Mode - NoteBooks / PySpark Shell**
2. **Bundle the code into a Jar and use Spark Submit to run the spark job.**

Note : Every Spark Job has one Driver, Application Master acts like a driver which gets registered with the Resource Manager. If the Driver goes down, then the application crashes.



Two Modes in which Spark Runs-

1. **Client Mode -** It is an interactive mode where the intent is to view the results instantly. Notebooks / PySpark Shell are used for interactive mode.

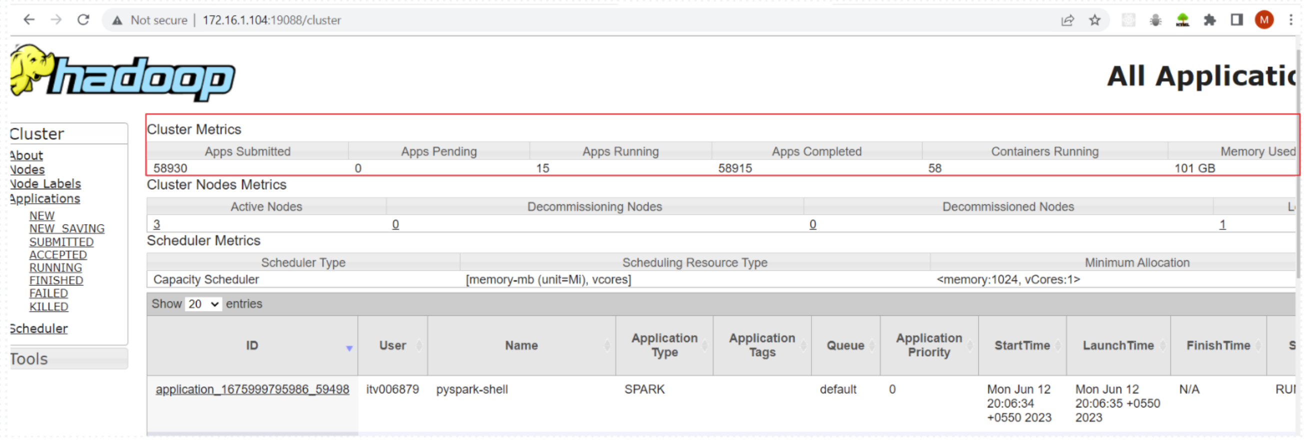
   Driver runs on the Client machine / Gateway node.

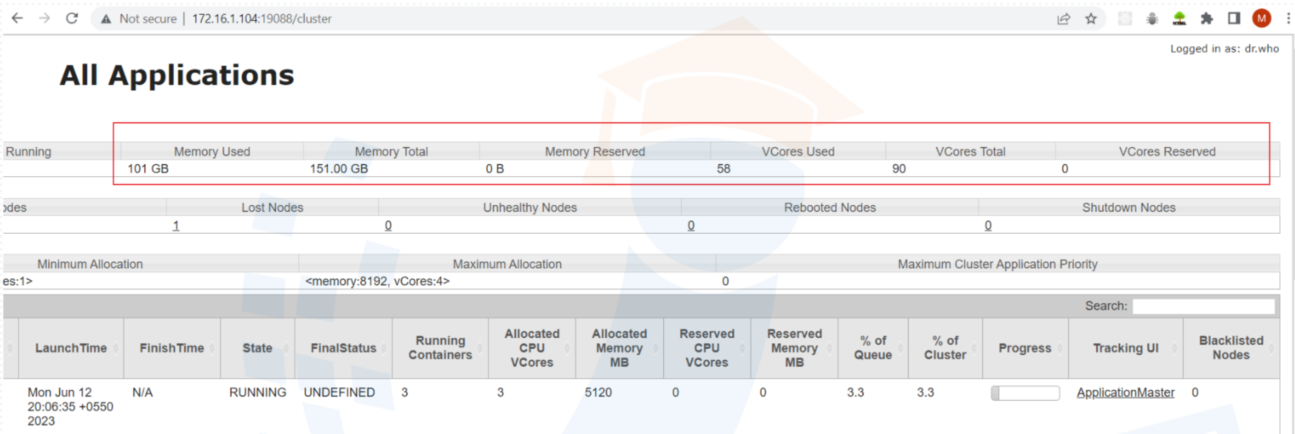
2. **Cluster Mode -** Code is packaged and submitted to the cluster for execution using Spark Submit.

   Driver runs on the Cluster. Recommended for Production environment.

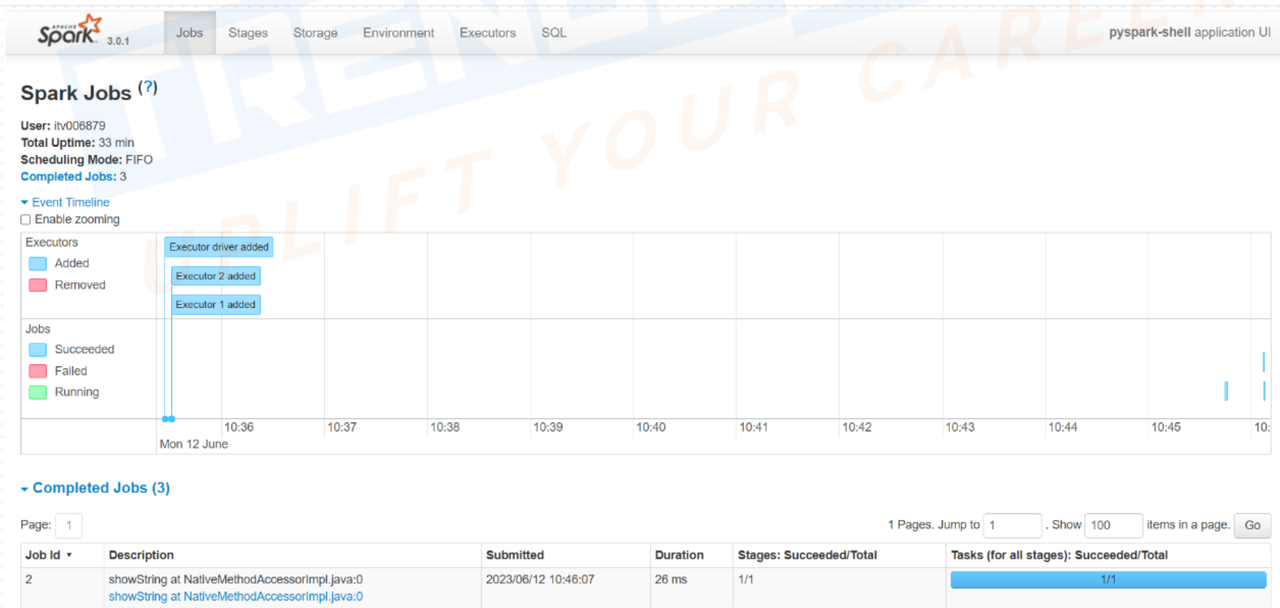
Resource Manager UI

Provides detailed information of the applications running and resources used like Application status, Memory, VCores…


Not secure | 172.16.1.104:19088/cluster
hadoop
All Applicatio
Cluster
About
Nodes
Node Labels
Applications
NEW
NEW_SAVING
SUBMITTED
ACCEPTED
RUNNING
FINISHED
FAILED
KILLED
Scheduler
Tools
Cluster Metrics
Apps Submitted | Apps Pending | Apps Running | Apps Completed | Containers Running | Memory Used
58930 | 0 | 15 | 58915 | 58 | 101 GB
Cluster Nodes Metrics
Active Nodes | Decommissioning Nodes | Decommissioned Nodes
3 | 0 | 0 | 1
Scheduler Metrics
Scheduler Type | Scheduling Resource Type | Minimum Allocation
Capacity Scheduler | [memory-mb (unit=Mi), vcores] | <memory:1024, vCores:1>
Show 20 entries
ID | User | Name | Application Type | Application Tags | Queue | Application Priority | StartTime | LaunchTime | FinishTime
application_1675999795986_59498 | itv006879 | pyspark-shell | SPARK | default | 0 | Mon Jun 12 20:06:34 +0550 2023 | Mon Jun 12 20:06:35 +0550 2023 | N/A


Not secure | 172.16.1.104:19088/cluster
Logged in as: dr.who
All Applications
Running
Memory Used | Memory Total | Memory Reserved | VCores Used | VCores Total | VCores Reserved
101 GB | 151.00 GB | 0 B | 58 | 90 | 0
Lost Nodes | Unhealthy Nodes | Rebooted Nodes | Shutdown Nodes
1 | 0 | 0 | 0
Minimum Allocation | Maximum Allocation | Maximum Cluster Application Priority
es:1> | <memory:8192, vCores:4> | 0
Search:
LaunchTime | FinishTime | State | FinalStatus | Running Containers | Allocated CPU VCores | Allocated Memory MB | Reserved CPU VCores | Reserved Memory MB | % of Queue | % of Cluster | Progress | Tracking UI | Blacklisted Nodes
Mon Jun 12 20:06:35 +0550 2023 | N/A | RUNNING | UNDEFINED | 3 | 3 | 5120 | 0 | 0 | 3.3 | 3.3 | | ApplicationMaster | 0

Spark UI


Spark 3.0.1
Jobs
Stages
Storage
Environment
Executors
SQL
pyspark-shell application UI
Spark Jobs (?)
User: itv006879
Total Uptime: 33 min
Scheduling Mode: FIFO
Completed Jobs: 3
Event Timeline
Enable zooming
Executors
Added
Removed
Executor driver added
Executor 2 added
Executor 1 added
Jobs
Succeeded
Failed
Running
10:36
10:37
10:38
10:39
10:40
10:41
10:42
10:43
10:44
10:45
Mon 12 June
Completed Jobs (3)
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
2 | showString at NativeMethodAccessorImpl.java:0 showString at NativeMethodAccessorImpl.java:0 | 2023/06/12 10:46:07 | 26 ms | 1/1 | 1/1

Accessing Columns in PySpark

1. **String Notation**

    Ex : df.select("*").show()

2. **Prefixing Column name with Dataframe name**

    Ex : df.order_date

3. **Array Notation**

    Ex : df['order_date']

4. **Column Object Notation**

    Ex : column('order_date') **or** col('order_date')

5. **Column Expression**

    Ex : expr("order_date")

Why are there so many ways of accessing columns?

- **Prefixing Column name with Dataframe name**

If two different dataframes have columns with the same name, then this would lead to ambiguity as the system is not aware of which dataframe column to choose.

Ex - Say cust_id is the column available in both orders and customer dataframes. Then the system is not sure whether to pick cust_id of customer dataframe or orders dataframe. Therefore, prefix the dataframe name with the column name to prevent any ambiguity.

- **Column Expression**

Is required when evaluation needs to be performed in a SQL way.

Ex - Say you need to increment customer ID and create a new customer ID

expr("cust_id + 1 as new_cust_id")

- **Column Object**

Provides various predefined functions to achieve the desired results in a programmatic approach.

Ex -
orders_df.select(“*”).where(col(‘order_status’).like(‘PENDING%’).show()

OR

orders_df.select(“*”).where(“order_status like ‘PENDING%’ ”).show()

Aggregate Functions

Combining multiple input rows together to give a consolidated output.

Simple Aggregations Example

Consider you have a orders.csv dataset and you are required to

- Count the total number of records
- Count number of distinct invoice ids
- Sum of Quantities
- Average unit price

Lets see 3 ways of solving the above

1. **Programmatic style**

Create and load a dataframe

```
from pyspark.sql.functions import *

orders_df.select(count(“*”).alias(“row_count”),
countDistinct(“invoiceno”).alias(“unique_invoice”),
sum(“quantity”).alias(“total_quantity”),
avg(“unitprice”).alias(“avg_price”)).show()
```

2. **Column Expression style**

orders_df.selectExpr("count(*) as row_count", "count(distinct(invoiceno)) as unique_invoice", "sum(quantity) as total_quantity", "avg(unitprice) as avg_price").show()

3. **Spark SQL style**

Create a Orders table

spark.sql("select count(*) as row_count, count(distinct(invoiceno)) as unique_invoice, sum(quantity) as total_quantity, avg(unitprice) as avg_price from orders").show()

Grouping Aggregations Example

Consider you have a orders.csv dataset and you are required to group based on invoice number and country.

- Find the total quantity for each group
- Find the total invoice amount (Amount = Quantity * UnitPrice)

Lets see 3 ways of solving the above

1. **Programmatic style**

Create and load dataframe with the given dataset

```
from pyspark.sql.functions import *

summary_df = orders_df \
.groupBy("country","invoiceno") \
.agg(sum("quantity".alias("total_quantity"), sum(expr("quantity * unitprice")).alias("invoice_value")).sort("invoiceno")
```

2. **Column Expression style**

```
summary_df = orders_df \
.groupBy("country","invoiceno") \
.agg(expr("sum(quantity) as total_quantity"), expr("sum(quantity *
unitprice) as invoice_value")).sort("invoiceno")
```

3. **Spark SQL style**

Create a Orders table

```
orders_df.createOrReplaceTempView("orders")

spark.sql(""" select country, invoiceno, sum(quantity) as
total_quantity, sum(quantity * unitprice) as invoice_value from
orders group by country, invoiceno order by invoiceno """).show()
```

Windowing Aggregations Example

Consider you have a windowdata.csv under /public/trendytech/datasets/windowdata.csv in the lab dataset and you are required to define the following 3 parameters -

1. **Partition Column** Partition by based on country
2. **Sorting Column** Sort based on week number
3. **Window Size** (Define the size by mentioning the start row and end row)

- Find the running total of invoice value

Create and load dataframe with the given dataset

```
from pyspark.sql.functions import *

mywindow = Window.partitionBy("country") \
.orderBy("weeknum") \
```

```
.rowsBetween(Window.unboundedPreceding, Window.currentRow)

result_df =
orders_df.withColumn("running_total",sum("invoicevalue")
.over(mywindow))
result_df.show()
```

Simple Aggregations	Grouping Aggregations	Windowing Aggregations
Accepts multiple input rows to give only one output row	Accepts multiple input rows belonging to a group (grouping is done on a column). For each group, there is one output row	Output generated by performing operations on a predefined rows set within a window
SUM , COUNT, DISTINCT	groupBy	partitionBy, orderBy, rowsBetween

Windowing Functions

1. **rank**
2. **dense_rank**
3. **row_number**
4. **lead**
5. **lag**

Consider you have a windowdatamodified.csv under **/public/trendytech/datasets/windowdatamodified.csv** in the lab.

This csv file has some values repeating to demonstrate the behaviour of the above windowing functions.

Rank

```
from pyspark.sql import SparkSession
import getpass
username = getpass.getuser()
spark= SparkSession. \
builder. \
config('spark.ui.port','0'). \
config("spark.sql.warehouse.dir", f"/user/{username}/warehouse"). \
enableHiveSupport(). \
master('yarn'). \
getOrCreate()
```

```
orders_df = spark.read \
.format("csv") \
.option("inferSchema","true") \
.option("header","true") \
.load("/public/trendytech/datasets/windowdatamodified.csv")
```

```
from pyspark.sql import *
from pyspark.sql.functions import *
```

```
mywindow = Window.partitionBy("country") \
.orderBy(desc("invoicevalue"))
```

```
results_df = orders_df.withColumn("rank",rank().over(mywindow))
```

```
results_df.show()
```

```
+-------+-------+-----------+-------------+------------+----+
|country|weeknum|numinvoices|totalquantity|invoicevalue|rank|
+-------+-------+-----------+-------------+------------+----+
| Sweden|     50|          3|         3714|      2646.3|   1|
|Germany|     49|         12|         1852|      1800.0|   1|
|Germany|     50|         15|         1973|      1800.0|   1|
|Germany|     48|         11|         1795|      1600.0|   3|
|Germany|     51|          5|         1103|      1600.0|   3|
| France|     50|          6|          529|      537.32|   1|
| France|     51|          5|          847|       500.0|   2|
| France|     49|          9|         2303|       500.0|   2|
| France|     48|          4|         1299|       500.0|   2|
|Belgium|     48|          1|          528|       800.0|   1|
|Belgium|     51|          2|          942|       800.0|   1|
|Belgium|     50|          2|          285|      625.16|   3|
|Finland|     50|          1|         1254|       892.8|   1|
|  India|     49|          5|         1280|      3284.1|   1|
|  India|     50|          5|         1184|     2321.78|   2|
|  India|     51|          5|           95|       300.0|   3|
|  India|     48|          7|         2822|       300.0|   3|
|  Italy|     48|          1|          164|       427.8|   1|
|  Italy|     51|          1|          131|       383.7|   2|
|  Italy|     49|          1|           -2|       -17.0|   3|
+-------+-------+-----------+-------------+------------+----+
only showing top 20 rows
```

Dense Rank

```
results_df = orders_df.withColumn("rank",dense_rank().over(mywindow))
```

```
results_df.show()
```

```
+-------+-------+-----------+-------------+------------+----+
|country|weeknum|numinvoices|totalquantity|invoicevalue|rank|
+-------+-------+-----------+-------------+------------+----+
| Sweden|     50|          3|         3714|      2646.3|   1|
|Germany|     49|         12|         1852|      1800.0|   1|
|Germany|     50|         15|         1973|      1800.0|   1|
|Germany|     48|         11|         1795|      1600.0|   2|
|Germany|     51|          5|         1103|      1600.0|   2|
| France|     50|          6|          529|      537.32|   1|
| France|     51|          5|          847|       500.0|   2|
| France|     49|          9|         2303|       500.0|   2|
| France|     48|          4|         1299|       500.0|   2|
|Belgium|     48|          1|          528|       800.0|   1|
|Belgium|     51|          2|          942|       800.0|   1|
|Belgium|     50|          2|          285|      625.16|   2|
|Finland|     50|          1|         1254|       892.8|   1|
|  India|     49|          5|         1280|      3284.1|   1|
|  India|     50|          5|         1184|     2321.78|   2|
|  India|     51|          5|           95|       300.0|   3|
|  India|     48|          7|         2822|       300.0|   3|
|  Italy|     48|          1|          164|       427.8|   1|
|  Italy|     51|          1|          131|       383.7|   2|
|  Italy|     49|          1|           -2|       -17.0|   3|
+-------+-------+-----------+-------------+------------+----+
only showing top 20 rows
```

Note :

In Rank, some ranks can be skipped if there are clashes in the ranks.

In Dense Rank, the ranks are not skipped even if there are clashes in the ranks.

In Row Number, different row numbers are assigned even in case of a tie. It plays an important role in calculating the top-n results.

Consider an Example Scenario of how ranks are assigned to the students based on the marks scored

Student Name	Marks Scored	rank	dense-rank	row-number
Ankur	100	1	1	1
Satish	100	1	1	2
Kapil	100	1	1	3
Kaushik	99	4	2	4
Ram	99	4	2	5
Rohit	98	6	3	6

Row Number

```
results_df = orders_df.withColumn("rank",row_number().over(mywindow))
```

```
results_df.show()
```

```
+-------+-------+-----------+-------------+------------+----+
|country|weeknum|numinvoices|totalquantity|invoicevalue|rank|
+-------+-------+-----------+-------------+------------+----+
| Sweden|     50|          3|         3714|      2646.3|   1|
|Germany|     49|         12|         1852|      1800.0|   1|
|Germany|     50|         15|         1973|      1800.0|   2|
|Germany|     48|         11|         1795|      1600.0|   3|
|Germany|     51|          5|         1103|      1600.0|   4|
| France|     50|          6|          529|      537.32|   1|
| France|     51|          5|          847|       500.0|   2|
| France|     49|          9|         2303|       500.0|   3|
| France|     48|          4|         1299|       500.0|   4|
|Belgium|     48|          1|          528|       800.0|   1|
|Belgium|     51|          2|          942|       800.0|   2|
|Belgium|     50|          2|          285|      625.16|   3|
|Finland|     50|          1|         1254|       892.8|   1|
|  India|     49|          5|         1280|      3284.1|   1|
|  India|     50|          5|         1184|     2321.78|   2|
|  India|     51|          5|           95|       300.0|   3|
|  India|     48|          7|         2822|       300.0|   4|
|  Italy|     48|          1|          164|       427.8|   1|
|  Italy|     51|          1|          131|       383.7|   2|
|  Italy|     49|          1|           -2|       -17.0|   3|
+-------+-------+-----------+-------------+------------+----+
only showing top 20 rows
```

Note : When you need to compare two rows, then the lead or lag function is used.

Lead - Is used when the current row needs to be compared with the next row.

Lag - Is used when the current row needs to be compared with the previous row.

```
from pyspark.sql import SparkSession
import getpass
username = getpass.getuser()
spark= SparkSession. \
builder. \
config('spark.ui.port','0'). \
config("spark.sql.warehouse.dir", f"/user/{username}/warehouse"). \
enableHiveSupport(). \
master('yarn'). \
getOrCreate()
```

```
orders_df = spark.read \
.format("csv") \
.option("inferSchema","true") \
.option("header","true") \
.load("/public/trendytech/datasets/windowdatamodified.csv")
```

```
from pyspark.sql import *
from pyspark.sql.functions import *
```

```
mywindow = Window.partitionBy("country") \
.orderBy("weeknum")
```

```
results_df = orders_df.withColumn("previous_week",lag("invoicevalue").over(mywindow))
```

```
results_df.show()
```

```
+-------+-------+-----------+-------------+------------+-------------+
|country|weeknum|numinvoices|totalquantity|invoicevalue|previous_week|
+-------+-------+-----------+-------------+------------+-------------+
| Sweden|     50|          3|         3714|      2646.3|         null|
|Germany|     48|         11|         1795|      1600.0|         null|
|Germany|     49|         12|         1852|      1800.0|       1600.0|
|Germany|     50|         15|         1973|      1800.0|       1800.0|
|Germany|     51|          5|         1103|      1600.0|       1800.0|
| France|     48|          4|         1299|       500.0|         null|
| France|     49|          9|         2303|       500.0|        500.0|
| France|     50|          6|          529|      537.32|        500.0|
| France|     51|          5|          847|       500.0|       537.32|
|Belgium|     48|          1|          528|       800.0|         null|
|Belgium|     50|          2|          285|      625.16|        800.0|
|Belgium|     51|          2|          942|       800.0|       625.16|
|Finland|     50|          1|         1254|       892.8|         null|
|  India|     48|          7|         2822|       300.0|         null|
|  India|     49|          5|         1280|      3284.1|        300.0|
|  India|     50|          5|         1184|     2321.78|       3284.1|
|  India|     51|          5|           95|       300.0|      2321.78|
|  Italy|     48|          1|          164|       427.8|         null|
|  Italy|     49|          1|           -2|       -17.0|        427.8|
|  Italy|     51|          1|          131|       383.7|        -17.0|
+-------+-------+-----------+-------------+------------+-------------+
only showing top 20 rows
```

Analysing log files to find some valuable Inferences.

1. First develop the logic with sample data and then apply to original data.

//Create a Spark Session

logs_data = [("INFO","2015-8-8 20:49:22"),
("WARN","2015-1-14 20:05:00"),
("INFO","2017-6-14 00:08:35"),
("INFO","2016-1-18 11:50:14"),
("DEBUG","2017-7-1 12:55:02"),
("INFO","2014-2-26 12:34:21"),
("INFO","2015-7-12 11:13:47"),
("INFO","2017-4-15 01:20:18"),
("DEBUG","2016-11-2 20:19:23"),
("INFO","2012-8-20 10:09:44")] //Sample log Data

log_df = spark.createDataFrame(logs_data).toDF('log_level','log_time')
//creating dataframe from local hard-coded data

new_log_df = log_df.withColumn("logtime", to_timestamp("log_time"))
//changing the datatype of the column log_time from string to timestamp

//In order to operate on the data like a sql table create a temp view table.
new_log_df.createOrReplaceTempView("serverlogs")

spark.sql(select loglevel, date_format(logtime, 'MMMM') as month from serverlogs").show()

//if only the month needs to be extracted.

spark.sql(select loglevel, date_format(logtime, 'MMMM') as month, count(*) as total_occurence from serverlogs group by loglevel, month").show() //Applying transformations - total occurrence of different log status like WARN,INFO … in the log file by grouping based on loglevel and month

2. Now working on the original dataset after ensuring that the logic is functional without any errors.

Plug in the main dataset

logschema = "loglevel string, logtime timestamp"

**log_df = spark.read **
**.format("csv") **
**.schema(logschema) **
.load("/public/trendytech/datasets/logsdata1m.csv")

//Now the same aggregations as in the previous case can be applied to this dataframe.

Pivot Table

Provides a more intuitive view by which data can be easily analysed for insights

spark.sql("select loglevel, date_format(logtime, 'MMMM') as month from serverlogs").groupBy('loglevel').pivot('month').count.show()

Pivot Table View

Month ->

	January	February	March	April	May	June	●●●	December
Error	1000	1200						
Info	100	89						
Warn	100	99						
Fatal	100	100						
Debug	99	79						

Note :

Optimization - By explicitly providing the list of values on the pivot column, the system will not be scanning the data to create the list of pivot values. This will save some processing time and improve the performance of query execution.

month_list = ['Jan', 'Feb', 'Mar', … 'dec']

spark.sql("select loglevel, date_format(logtime, 'MMMM') as month from serverlogs").groupBy('loglevel').pivot('month', month_list).count.show()