Solution : Week 8 Assignment

Question 1: Research more on uber mode and create a detail.

Answer :

Uber Mode, the application is executed within the same container where the application master is running. The application master is responsible for coordinating the execution of tasks within a YARN application.

Uber Mode is primarily used for small jobs that don't require the resources of a full cluster. Instead of distributing the tasks across multiple containers in the cluster, Uber Mode allows the entire job to be executed within a single container, minimising the overhead associated with managing multiple containers and the associated inter-container communication.

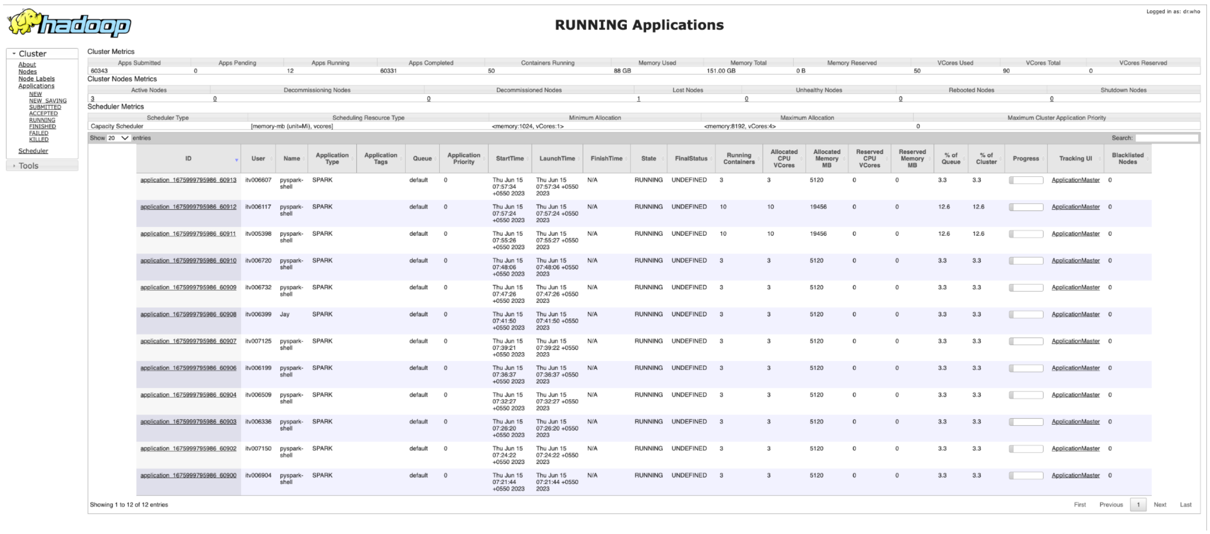
Question 2: Explore a multimode cluster by navigating to the resource manager in the labs.

Answer :
The Resource Manager UI allows users to monitor and manage the resources and applications running on a YARN cluster. It provides a graphical interface to view various aspects of the cluster, including cluster utilisation, running applications, and resource allocation.
Here are some key features and components of the Resource Manager UI:

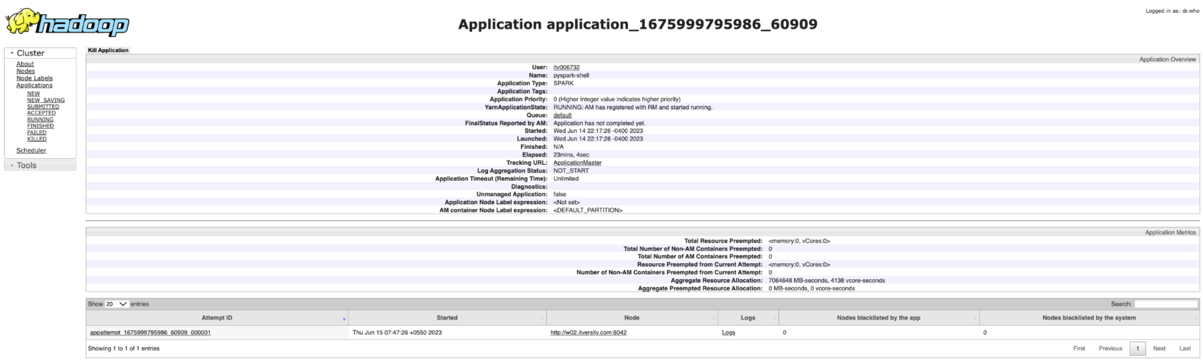
1. **Cluster Overview**: The UI provides an overview of the cluster, including information such as the number of active nodes, total and available memory, CPU utilization, and the number of running and completed applications.
2. **Application List**: The UI displays a list of applications running on the cluster. It includes details like the application ID, user, state (e.g., running, completed, failed), start and finish time, and resource usage.
3. **Application Details**: Clicking on an application in the application list opens a detailed view. This view provides more information about the selected application, such as the list of containers allocated to the application, their resource usage, log links, and diagnostics information.
4. **Cluster Metrics**: The UI presents graphical representations of cluster metrics over time. These metrics may include CPU utilisation, memory usage, and other resource utilisation statistics.
5. **Queue Information**: If the cluster has multiple queues configured for resource allocation, the UI displays information about the queues. This includes details like the queue name, capacity, maximum capacity, current usage, and pending applications.
6. **Node Information**: The UI provides a summary of the nodes in the cluster. It shows details about each node, such as node ID, state, health status, available and used resources, and the number of containers running on the node.
7. **Logs and Diagnostics**: The Resource Manager UI allows you to access the logs and diagnostics information for individual applications. This can be helpful for troubleshooting issues or investigating the performance of specific applications.

The Resource Manager UI is a powerful tool for monitoring and managing resources in a YARN cluster. It enables administrators and users to track the cluster's health, monitor application status, analyse resource usage, and diagnose problems. By leveraging the information provided by the UI, users can optimise resource allocation, identify bottlenecks, and ensure efficient cluster utilisation.

In the Resource Manager UI, you can see an overview of the applications, including the number of submitted and running applications. It also provides information about the memory and core usage out of the total available resources. Additionally, you can check the number of active nodes running in the cluster.
As a user with the ID "itv006732," I am currently running a Spark session. I will explore the same information in the Spark UI.



Once you click on the application ID of the application you are running, you will be redirected to a page where you can access more detailed information about your application. This page provides information about the status of your application, the tracking URL for your job, the start and finish times of your application, and the current state of the YARN application.



Question4 : How to deal with nulls in Apache Spark? Please provide a detailed explanation with examples where necessary

Answer:

Dealing with null values in Apache Spark is an essential task to ensure accurate and reliable data processing. Apache Spark provides several methods and functions to handle null values effectively. In this explanation, I'll provide a detailed overview of common techniques and examples for dealing with nulls in Spark.

Dropping Null Values:

The simplest approach is to remove rows containing null values from your dataset. Spark provides the na object that allows you to perform operations on missing data. The drop() function can be used to drop rows with null values.

Example:

```
from pyspark.sql import SparkSession
from pyspark.sql.functions import col
spark = SparkSession.builder.getOrCreate()
df = spark.read.option("header", "true").csv("data.csv")
cleanedDF = df.na.drop()
cleanedDF.show()
```

Filling Null Values:

Another approach is to fill the null values with specific default or derived values. Spark provides the fill() function to replace nulls with a specified value.

Example:

```
from pyspark.sql import SparkSession
from pyspark.sql.functions import col
spark = SparkSession.builder.getOrCreate()
df = spark.read.option("header", "true").csv("data.csv")
filledDF = df.na.fill("N/A")
filledDF.show()
```

Additionally, you can fill nulls with different values based on the data type of the column. For numeric columns, you can use the fill() function with a numeric value, and for boolean columns, you can use fill() with a Boolean value.

Replacing Nulls with Conditional Values:

You may want to replace nulls with specific values based on certain conditions. Spark provides the when() and otherwise() functions, which allow you to define conditional logic while replacing nulls.

Example:

```
from pyspark.sql import SparkSession
from pyspark.sql.functions import col, when
spark = SparkSession.builder.getOrCreate()
df = spark.read.option("header", "true").csv("data.csv")
replacedDF = df.withColumn("new_column",
                    when(col("column1").isNull(), "Unknown")
                    .otherwise(col("column1")))
replacedDF.show()
```

Filtering Null Values:

Sometimes, you may want to filter out rows containing null values for specific columns. You can use the filter() function along with isNull() or isNotNull() to achieve this.

Example:

```
from pyspark.sql import SparkSession
from pyspark.sql.functions import col
spark = SparkSession.builder.getOrCreate()
df = spark.read.option("header", "true").csv("data.csv")
filteredDF = df.filter(col("column1").isNotNull())
filteredDF.show()
```

Handling Nulls in Aggregations:

When performing aggregations on columns that contain nulls, Spark provides specialised functions such as coalesce(), first(), last(), and avg() that handle nulls appropriately.

Example:

```
from pyspark.sql import SparkSession
from pyspark.sql.functions import avg, coalesce, first, lit
spark = SparkSession.builder.getOrCreate()
df = spark.read.option("header", "true").csv("data.csv")
aggregatedDF = df.groupBy("group_column").agg(
        avg("numeric_column"),
        coalesce(first("column1"), lit("N/A")).alias("new_column")
    )
aggregatedDF.show()
```

These are some common techniques for dealing with null values in Apache Spark. The choice of method depends on the specific requirements of your data analysis or processing tasks. Remember to import the necessary functions from org.apache.spark.sql.functions to use the functions mentioned above.

Question 3: Select a relevant dataset of your choice and write some queries to demonstrate the following: running total, grouping aggregates, and various window functions like rank, dense_rank, row_number, lead, and lag. Also, try creating a pivot view.

Solution:

Link for code → https://g01.itversity.com/hub/user-redirect/lab/tree/q4_assignment.ipynb

Snapshot :

I am using hotel data to perform simple aggregations, such as counting. Here I am enforcing schema so we don't face any issue later with data type.

In [1]:
```
#create a first spark session here

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/itv006732/warehouse").\
enableHiveSupport(). \
master('yarn'). \
getOrCreate()
```

In [2]:
```
spark
```

Out[2]:

SparkSession - hive

SparkContext

Spark UI

Version

`v3.0.1`

Master

`yarn`

AppName

`pyspark-shell`

In [3]:
```
from pyspark.sql.types import StructType, StructField, StringType, LongType, DateType, IntegerType,FloatType


hotel_schema = StructType([
    StructField("customer_id", LongType(), True),
    StructField("customer_name", StringType(), True),
    StructField("check_in_date", DateType(), True),
    StructField("check_out_date", DateType(), True),
    StructField("room_type", StringType(), True),
    StructField("price", FloatType(), True)
])

# Read the file with the enforced schema
#let's use hotel_data for doing basis analysis
hotel_df = spark.read \
    .format("csv") \
    .option("header", "true") \
    .schema(hotel_schema) \
    .load("/public/trendytech/datasets/hotel_data.csv")
#/public/trendytech/datasets/train.csv
#/public/trendytech/datasets/hospital.csv
#/public/trendytech/datasets/order_data.csv


hotel_df.printSchema()
```

```
root
 |-- customer_id: long (nullable = true)
 |-- customer_name: string (nullable = true)
 |-- check_in_date: date (nullable = true)
 |-- check_out_date: date (nullable = true)
 |-- room_type: string (nullable = true)
 |-- price: float (nullable = true)
```

In [4]:
```
hotel_df.show()
```

```
+-----------+-----------------+-------------+--------------+---------+-----+
|customer_id|    customer_name|check_in_date|check_out_date|room_type|price|
+-----------+-----------------+-------------+--------------+---------+-----+
|          2|       Jane Smith|   2023-05-02|    2023-05-06|   Deluxe|600.0|
|          3|     Mark Johnson|   2023-05-03|    2023-05-08| Standard|450.0|
|          4|     Sarah Wilson|   2023-05-04|    2023-05-07|Executive|750.0|
|          5|      Emily Brown|   2023-05-06|    2023-05-09|   Deluxe|550.0|
|          6|    Michael Davis|   2023-05-07|    2023-05-10| Standard|400.0|
|          7|Samantha Thompson|   2023-05-08|    2023-05-12|   Deluxe|600.0|
|          8|     William Lee|   2023-05-10|    2023-05-13| Standard|450.0|
|          9|    Amanda Harris|   2023-05-11|    2023-05-16|Executive|750.0|
|         10|  David Rodriguez|   2023-05-12|    2023-05-15|   Deluxe|550.0|
|         11|     Linda Wilson|   2023-05-14|    2023-05-18| Standard|400.0|
|         12|   Robert Johnson|   2023-05-15|    2023-05-20|   Deluxe|600.0|
|         13|  Sophia Anderson|   2023-05-16|    2023-05-21| Standard|450.0|
|         14|      James Smith|   2023-05-17|    2023-05-23|Executive|750.0|
|         15|     Olivia Brown|   2023-05-19|    2023-05-24|   Deluxe|550.0|
|         16|    Michael Davis|   2023-05-20|    2023-05-25| Standard|400.0|
|         17|   Emily Thompson|   2023-05-21|    2023-05-27|   Deluxe|600.0|
|         18|      William Lee|   2023-05-23|    2023-05-28| Standard|450.0|
|         19|       Ava Harris|   2023-05-24|    2023-05-30|Executive|750.0|
|         20| Daniel Rodriguez|   2023-05-25|    2023-05-29|   Deluxe|550.0|
|         21|    Sophia Wilson|   2023-05-27|    2023-06-01| Standard|400.0|
+-----------+-----------------+-------------+--------------+---------+-----+
only showing top 20 rows
```

In [5]:
```
#let run the below command to use all sql functions in pyspark3

from pyspark.sql.functions import *
```

Simple aggregations:

In [6]:
```
#Let's find the number of records in this table.
hotel_df.select(count("*").alias("row_count")).show()
```

```
+---------+
|row_count|
+---------+
|      106|
+---------+
```

In [7]:
```
# Let's find the number of unique customers for each room type.
summary_df=hotel_df.groupBy("room_type").agg(count("customer_id").alias("number_of_customer")).sort("room_type")
summary_df.show()
```

```
+---------+------------------+
|room_type|number_of_customer|
+---------+------------------+
|   Deluxe|                43|
|Executive|                20|
| Standard|                43|
+---------+------------------+
```

In [8]:
```
# Create new session use hospital data set for the to perform more aggregation operation
```

In [9]:
```
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/itv006732/warehouse").\
enableHiveSupport(). \
master('yarn'). \
getOrCreate()
```

In [8]:
```
# Create new session use hospital data set for the to perform more aggregation operation
```

In [9]:
```
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/itv006732/warehouse").\
enableHiveSupport(). \
master('yarn'). \
getOrCreate()
```

In [10]:
```
spark
```

Out[10]:

SparkSession - hive

SparkContext

Spark UI

Version

`v3.0.1`

Master

`yarn`

AppName

`pyspark-shell`

Grouping aggregations

In [11]:
```


hospital_df = spark.read \
    .format("csv") \
    .option("header", "true") \
    .option("inferschema","true") \
    .load("/public/trendytech/datasets/hospital.csv")

hospital_df.printSchema()
```

```
root
 |-- patient_id: integer (nullable = true)
 |-- admission_date: string (nullable = true)
 |-- discharge_date: string (nullable = true)
 |-- diagnosis: string (nullable = true)
 |-- doctor_id: integer (nullable = true)
 |-- total_cost: double (nullable = true)
```

In [12]:
```
hospital_df.show()
```

```
+----------+--------------+--------------+-------------+---------+----------+
|patient_id|admission_date|discharge_date|    diagnosis|doctor_id|total_cost|
+----------+--------------+--------------+-------------+---------+----------+
|         1|    01-01-2022|    2022-01-10|    Pneumonia|      101|    5000.0|
|         2|    02-05-2022|    2022-02-09| Appendicitis|      102|    7000.0|
|         3|    03-12-2022|    2022-03-18|Fractured Arm|      103|    3500.0|
|         4|    04-02-2022|    2022-04-08| Heart Attack|      104|   15000.0|
|         5|    05-05-2022|    2022-05-07|    Influenza|      105|    2500.0|
|         6|    06-10-2022|    2022-06-15| Appendicitis|      106|    8000.0|
|         7|    07-20-2022|    2022-07-25|    Pneumonia|      107|    5500.0|
|         8|    08-25-2022|    2022-09-01| Heart Attack|      108|   20000.0|
|         9|    09-15-2022|    2022-09-22|Fractured Leg|      109|    6000.0|
|        10|    10-05-2022|    2022-10-10| Appendicitis|      110|    7500.0|
|        11|    11-02-2022|    2022-11-05|    Influenza|      111|    2800.0|
|        12|    12-10-2022|    2022-12-18|    Pneumonia|      112|    6000.0|
|        13|    01-02-2023|    2023-01-09| Heart Attack|      113|   18000.0|
|        14|    02-14-2023|    2023-02-18| Appendicitis|      114|    7200.0|
|        15|    03-20-2023|    2023-03-28|Fractured Arm|      115|    3800.0|
|        16|    04-05-2023|    2023-04-11|    Influenza|      116|    2700.0|
|        17|    05-08-2023|    2023-05-11| Heart Attack|      117|   16000.0|
```

In [13]:
```
from pyspark.sql.functions import *
```

In [14]:
```
#Let's convert the string to a datetime column for performing operations.
result_df1 = hospital_df.withColumn("admission_date", to_date("admission_date", "dd-mm-yyyy"))
result_df2 = result_df1.withColumn("discharge_date", to_date("discharge_date", "yyyy-mm-dd"))
```

In [15]:
```
result_df2.show()
```

```
+----------+--------------+--------------+-------------+---------+----------+
|patient_id|admission_date|discharge_date|    diagnosis|doctor_id|total_cost|
+----------+--------------+--------------+-------------+---------+----------+
|         1|    2022-01-01|    2022-01-10|    Pneumonia|      101|    5000.0|
|         2|    2022-01-02|    2022-01-09| Appendicitis|      102|    7000.0|
|         3|    2022-01-03|    2022-01-18|Fractured Arm|      103|    3500.0|
|         4|    2022-01-04|    2022-01-08| Heart Attack|      104|   15000.0|
|         5|    2022-01-05|    2022-01-07|    Influenza|      105|    2500.0|
|         6|    2022-01-06|    2022-01-15| Appendicitis|      106|    8000.0|
|         7|    2022-01-07|    2022-01-25|    Pneumonia|      107|    5500.0|
|         8|    2022-01-08|    2022-01-01| Heart Attack|      108|   20000.0|
|         9|    2022-01-09|    2022-01-22|Fractured Leg|      109|    6000.0|
|        10|    2022-01-10|    2022-01-10| Appendicitis|      110|    7500.0|
|        11|    2022-01-11|    2022-01-05|    Influenza|      111|    2800.0|
|        12|    2022-01-12|    2022-01-18|    Pneumonia|      112|    6000.0|
|        13|    2023-01-01|    2023-01-09| Heart Attack|      113|   18000.0|
|        14|    2023-01-02|    2023-01-18| Appendicitis|      114|    7200.0|
|        15|    2023-01-03|    2023-01-28|Fractured Arm|      115|    3800.0|
|        16|    2023-01-04|    2023-01-11|    Influenza|      116|    2700.0|
|        17|    2023-01-05|    2023-01-11| Heart Attack|      117|   16000.0|
|        18|    2023-01-06|    2023-01-20|    Pneumonia|      118|    4800.0|
|        19|    2023-01-07|    2023-01-27|Fractured Leg|      119|    6500.0|
|        20|    2023-01-08|    2023-01-16| Appendicitis|      120|    7800.0|
+----------+--------------+--------------+-------------+---------+----------+
only showing top 20 rows
```

In [16]:
```
# Let's create a new column, 'year', which represents the year in which the patients are getting admitted."

result_df3=result_df2.withColumn("year",year("admission_date"))
```

In [17]:
```
result_df3.show()
```

```
+----------+--------------+--------------+-------------+---------+----------+----+
|patient_id|admission_date|discharge_date|    diagnosis|doctor_id|total_cost|year|
+----------+--------------+--------------+-------------+---------+----------+----+
|         1|    2022-01-01|    2022-01-10|    Pneumonia|      101|    5000.0|2022|
|         2|    2022-01-02|    2022-01-09| Appendicitis|      102|    7000.0|2022|
|         3|    2022-01-03|    2022-01-18|Fractured Arm|      103|    3500.0|2022|
|         4|    2022-01-04|    2022-01-08| Heart Attack|      104|   15000.0|2022|
|         5|    2022-01-05|    2022-01-07|    Influenza|      105|    2500.0|2022|
|         6|    2022-01-06|    2022-01-15| Appendicitis|      106|    8000.0|2022|
|         7|    2022-01-07|    2022-01-25|    Pneumonia|      107|    5500.0|2022|
```

In [18]:
```
# Let's find the number of patients diagnosed in each year, categorized by different categories.
summary_df=result_df3.groupby("year","diagnosis").agg(count("patient_id").alias("total_patient")).sort(desc("year"))
summary_df.show()
```

```
+----+-------------+-------------+
|year|    diagnosis|total_patient|
+----+-------------+-------------+
|2024|    Influenza|            1|
|2023|    Influenza|            2|
|2023| Appendicitis|            2|
|2023|    Pneumonia|            2|
|2023| Heart Attack|            3|
|2023|Fractured Leg|            1|
|2023|Fractured Arm|            2|
|2022| Appendicitis|            3|
|2022|Fractured Leg|            1|
|2022|    Pneumonia|            3|
|2022|    Influenza|            2|
|2022| Heart Attack|            2|
|2022|Fractured Arm|            1|
+----+-------------+-------------+
```

In [19]:
```
#For better visualization, let's create a pivot table based on this data and replace any null values with 0

summary_df.groupBy("diagnosis").pivot("year").count().na.fill(0).show()
```

```
+-------------+----+----+----+
|    diagnosis|2022|2023|2024|
+-------------+----+----+----+
| Heart Attack|   1|   1|   0|
|Fractured Arm|   1|   1|   0|
|Fractured Leg|   1|   1|   0|
| Appendicitis|   1|   1|   0|
|    Influenza|   1|   1|   1|
|    Pneumonia|   1|   1|   0|
+-------------+----+----+----+
```

In [20]:
```
# Let's find the total number of patients for each category
from pyspark.sql import Window
from pyspark.sql.functions import desc

my_window=Window.partitionBy("diagnosis")
```

In [21]:
```
summary_df2=result_df3.withColumn("total_patient_each_category",count("patient_id").over(my_window))
```

In [22]:
```
summary_df2.show()
```

```
+----------+--------------+--------------+-------------+---------+----------+----+---------------------------+
|patient_id|admission_date|discharge_date|    diagnosis|doctor_id|total_cost|year|total_patient_each_category|
+----------+--------------+--------------+-------------+---------+----------+----+---------------------------+
|         4|    2022-01-04|    2022-01-08| Heart Attack|      104|   15000.0|2022|                          5|
|         8|    2022-01-08|    2022-01-01| Heart Attack|      108|   20000.0|2022|                          5|
|        13|    2023-01-01|    2023-01-09| Heart Attack|      113|   18000.0|2023|                          5|
|        17|    2023-01-05|    2023-01-11| Heart Attack|      117|   16000.0|2023|                          5|
|        22|    2023-01-10|    2023-01-19| Heart Attack|      122|   21000.0|2023|                          5|
|         3|    2022-01-03|    2022-01-18|Fractured Arm|      103|    3500.0|2022|                          3|
|        15|    2023-01-03|    2023-01-28|Fractured Arm|      115|    3800.0|2023|                          3|
|        24|    2023-01-12|    2023-01-07|Fractured Arm|      124|    4100.0|2023|                          3|
|         9|    2022-01-09|    2022-01-22|Fractured Leg|      109|    6000.0|2022|                          2|
|        19|    2023-01-07|    2023-01-27|Fractured Leg|      119|    6500.0|2023|                          2|
|         2|    2022-01-02|    2022-01-09| Appendicitis|      102|    7000.0|2022|                          5|
|         6|    2022-01-06|    2022-01-15| Appendicitis|      106|    8000.0|2022|                          5|
|        10|    2022-01-10|    2022-01-10| Appendicitis|      110|    7500.0|2022|                          5|
|        14|    2023-01-02|    2023-01-18| Appendicitis|      114|    7200.0|2023|                          5|
|        20|    2023-01-08|    2023-01-16| Appendicitis|      120|    7800.0|2023|                          5|
|         5|    2022-01-05|    2022-01-07|    Influenza|      105|    2500.0|2022|                          5|
|        11|    2022-01-11|    2022-01-05|    Influenza|      111|    2800.0|2022|                          5|
|        16|    2023-01-04|    2023-01-11|    Influenza|      116|    2700.0|2023|                          5|
|        21|    2023-01-09|    2023-01-09|    Influenza|      121|    2900.0|2023|                          5|
|        25|    2024-01-01|    2024-01-15|    Influenza|      125|    3200.0|2024|                          5|
+----------+--------------+--------------+-------------+---------+----------+----+---------------------------+
only showing top 20 rows
```

In [23]:
```


from pyspark.sql.functions import row_number
from pyspark.sql.window import Window

my_window2 = Window.partitionBy("diagnosis").orderBy(summary_df2["total_patient_each_category"].desc())

summary_df3 = summary_df2.withColumn("row_num", row_number().over(my_window2))

top3_df = summary_df3.filter(summary_df3["row_num"] <= 1)

#let's take neccessery column
top3_df['diagnosis','total_patient_each_category'].show()
```

```
+-------------+---------------------------+
|    diagnosis|total_patient_each_category|
+-------------+---------------------------+
| Heart Attack|                          5|
|Fractured Arm|                          3|
|Fractured Leg|                          2|
| Appendicitis|                          5|
|    Influenza|                          5|
|    Pneumonia|                          5|
+-------------+---------------------------+
```

In [24]:

```
#Let's find the top 3 patients who have the highest total cost for each diagnosis.

my_window3=Window.partitionBy("diagnosis").orderBy(desc("total_cost"))
summary_df4=result_df3.withColumn("rank",rank().over(my_window3))
top3_df=summary_df4.filter(summary_df4["rank"]<=3)
```

In [25]:

```
top3_df.show()
```

```
+----------+--------------+--------------+-------------+---------+----------+----+----+
|patient_id|admission_date|discharge_date|    diagnosis|doctor_id|total_cost|year|rank|
+----------+--------------+--------------+-------------+---------+----------+----+----+
|        22|    2023-01-10|    2023-01-19| Heart Attack|      122|   21000.0|2023|   1|
|         8|    2022-01-08|    2022-01-01| Heart Attack|      108|   20000.0|2022|   2|
|        13|    2023-01-01|    2023-01-09| Heart Attack|      113|   18000.0|2023|   3|
|        24|    2023-01-12|    2023-01-07|Fractured Arm|      124|    4100.0|2023|   1|
|        15|    2023-01-03|    2023-01-28|Fractured Arm|      115|    3800.0|2023|   2|
|         3|    2022-01-03|    2022-01-18|Fractured Arm|      103|    3500.0|2022|   3|
|        19|    2023-01-07|    2023-01-27|Fractured Leg|      119|    6500.0|2023|   1|
|         9|    2022-01-09|    2022-01-22|Fractured Leg|      109|    6000.0|2022|   2|
|         6|    2022-01-06|    2022-01-15| Appendicitis|      106|    8000.0|2022|   1|
|        20|    2023-01-08|    2023-01-16| Appendicitis|      120|    7800.0|2023|   2|
|        10|    2022-01-10|    2022-01-10| Appendicitis|      110|    7500.0|2022|   3|
|        25|    2024-01-01|    2024-01-15|    Influenza|      125|    3200.0|2024|   1|
|        21|    2023-01-09|    2023-01-09|    Influenza|      121|    2900.0|2023|   2|
|        11|    2022-01-11|    2022-01-05|    Influenza|      111|    2800.0|2022|   3|
|        12|    2022-01-12|    2022-01-18|    Pneumonia|      112|    6000.0|2022|   1|
|         7|    2022-01-07|    2022-01-25|    Pneumonia|      107|    5500.0|2022|   2|
|        23|    2023-01-11|    2023-01-22|    Pneumonia|      123|    5200.0|2023|   3|
+----------+--------------+--------------+-------------+---------+----------+----+----+
```

In [26]:

```
# let's drop the rank column there is no use as of now for this.

top3_df.drop("rank","year").show()
```

```
+----------+--------------+--------------+-------------+---------+----------+
|patient_id|admission_date|discharge_date|    diagnosis|doctor_id|total_cost|
+----------+--------------+--------------+-------------+---------+----------+
|        22|    2023-01-10|    2023-01-19| Heart Attack|      122|   21000.0|
|         8|    2022-01-08|    2022-01-01| Heart Attack|      108|   20000.0|
|        13|    2023-01-01|    2023-01-09| Heart Attack|      113|   18000.0|
|        24|    2023-01-12|    2023-01-07|Fractured Arm|      124|    4100.0|
|        15|    2023-01-03|    2023-01-28|Fractured Arm|      115|    3800.0|
|         3|    2022-01-03|    2022-01-18|Fractured Arm|      103|    3500.0|
|        19|    2023-01-07|    2023-01-27|Fractured Leg|      119|    6500.0|
|         9|    2022-01-09|    2022-01-22|Fractured Leg|      109|    6000.0|
|         6|    2022-01-06|    2022-01-15| Appendicitis|      106|    8000.0|
|        20|    2023-01-08|    2023-01-16| Appendicitis|      120|    7800.0|
|        10|    2022-01-10|    2022-01-10| Appendicitis|      110|    7500.0|
```