Let's delve into Partitioning and Bucketing in PySpark first, followed by Job Scheduling.

21. Partitioning and Bucketing:

In PySpark, partitioning and bucketing are techniques used to organize data for better performance.

a. Partitioning:
Partitioning involves dividing the data into partitions based on a specified column or expression. It helps in efficient data retrieval as operations can be performed on specific partitions, reducing the amount of data that needs to be processed.

b. Bucketing:
Bucketing involves dividing data into a fixed number of buckets based on a hash function. This can improve query performance, especially when used with partitioning. It ensures that similar data is stored together, making certain operations more efficient.

Sample PySpark Code:

```python
from pyspark.sql import SparkSession
from pyspark.sql.functions import col

# Create a Spark session
spark = SparkSession.builder.appName("PartitioningAndBucketing").getOrCreate()

# Sample DataFrame
data = [("Alice", 25), ("Bob", 30), ("Charlie", 22), ("David", 35)]
columns = ["Name", "Age"]
df = spark.createDataFrame(data, columns)

# Partitioning by the 'Age' column
df_partitioned = df.repartition("Age")

# Bucketing the partitioned DataFrame
df_bucketed = df_partitioned.write.bucketBy(3, "Age").saveAsTable("bucketed_table",
format="parquet", mode="overwrite")
```

In this example, we first create a DataFrame and then partition it based on the 'Age' column. After partitioning, we apply bucketing with 3 buckets on the 'Age' column and save the DataFrame as a table.

22. Job Scheduling:

Job scheduling in PySpark involves controlling parallelism and resource allocation.

a. Controlling Parallelism:
PySpark allows you to control the level of parallelism when performing operations. You can set the number of partitions for RDDs or control the number of tasks for DataFrames to optimize parallel execution.

b. Resource Allocation:
PySpark also provides configuration options to allocate resources efficiently. You can configure the number of executor instances, memory per executor, and other parameters to ensure optimal resource utilization.

Sample PySpark Code:

```python
from pyspark.sql import SparkSession

# Create a Spark session with custom configurations
spark = SparkSession.builder \
    .appName("JobSchedulingExample") \
    .config("spark.executor.instances", "2") \
    .config("spark.executor.memory", "2g") \
    .getOrCreate()

# Sample DataFrame
data = [("Alice", 25), ("Bob", 30), ("Charlie", 22), ("David", 35)]
columns = ["Name", "Age"]
df = spark.createDataFrame(data, columns)

# Perform operations with controlled parallelism
result = df.repartition(2).filter("Age > 25").show()
```

In this example, we create a Spark session with custom configurations to control the number of executor instances and memory per executor. We then perform operations on the DataFrame with controlled parallelism using `repartition`.