NIQ Interview Questions and Answers

Link to practice - Python and PySpark:
Link to practice - SQL practice:

1. What technologies are you strong in?

Answer: I have strong expertise in Python, PySpark, SQL, and data engineering tools. I am proficient in using Spark for big data processing, and I have experience with various data storage and processing frameworks such as Hadoop, Hive, and Delta Lake.

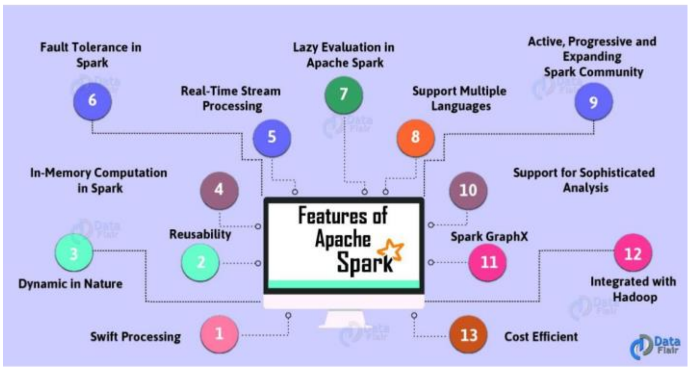
2. What is your total experience?

Answer: I have a total of X years of experience in the field of data engineering and analytics, with a focus on big data technologies and distributed computing.

3. Have you faced difficult issues/problems in Spark, and how did you deal with them?

Answer: Yes, I have encountered several challenging issues in Spark, such as performance bottlenecks and memory management problems. To address these, I have optimized Spark jobs by tuning configurations, using efficient data partitioning, and implementing caching and persistence strategies. Additionally, I have leveraged Spark's built-in tools for debugging and monitoring to identify and resolve issues.

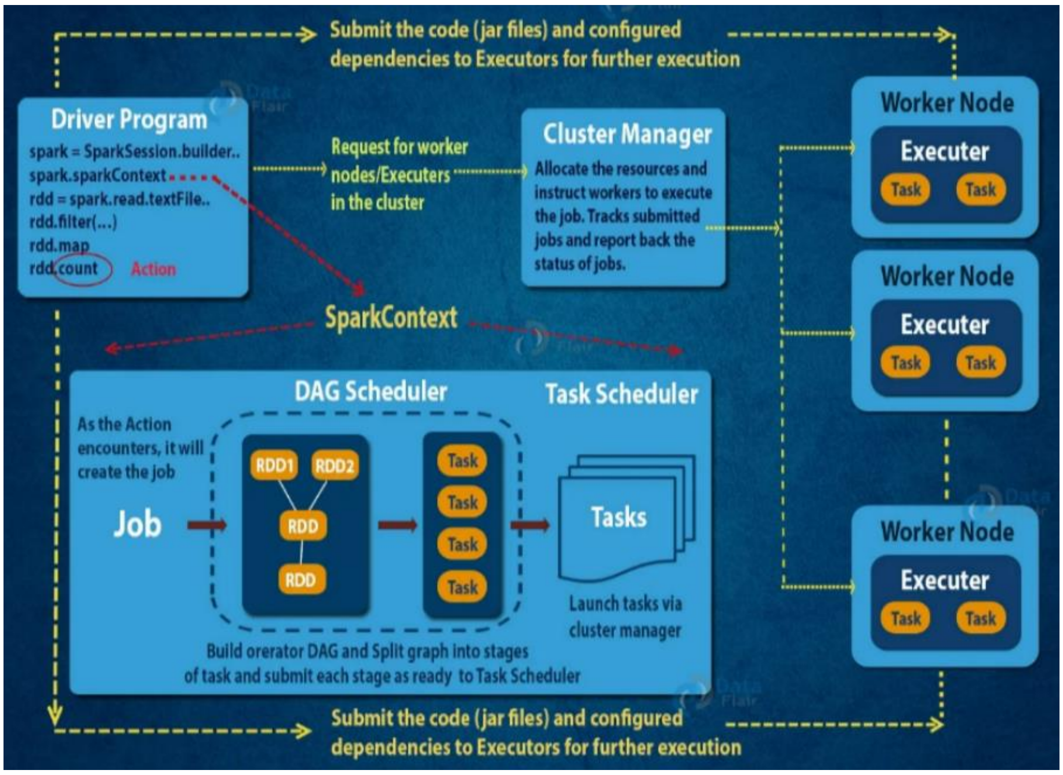
4. What are features of Spark?


Features of Apache Spark
1 Swift Processing
2 Reusability
3 Dynamic in Nature
4 In-Memory Computation in Spark
5 Real-Time Stream Processing
6 Fault Tolerance in Spark
7 Lazy Evaluation in Apache Spark
8 Support Multiple Languages
9 Active, Progressive and Expanding Spark Community
10 Support for Sophisticated Analysis
11 Spark GraphX
12 Integrated with Hadoop
13 Cost Efficient
Data Flair

5. Explain Spark Eco System.

Spark SQL
Spark Streaming
Spark MLlib
Spark GraphX
Spark R
Spark
Scala
R

6. How DAG works in Spark.


Submit the code (jar files) and configured dependencies to Executors for further execution
Driver Program
spark = SparkSession.builder..
spark.sparkContext
rdd = spark.read.textFile..
rdd.filter(...)
rdd.map
rdd.count
Action
Request for worker nodes/Executers in the cluster
Cluster Manager
Allocate the resources and instruct workers to execute the job. Tracks submitted jobs and report back the status of jobs.
Worker Node
Executer
Task
Task
Worker Node
Executer
Task
Task
SparkContext
DAG Scheduler
Task Scheduler
As the Action encounters, it will create the job
Job
RDD1
RDD2
RDD
RDD
Task
Task
Task
Task
Tasks
Launch tasks via cluster manager
Build orerator DAG and Split graph into stages of task and submit each stage as ready to Task Scheduler
Worker Node
Executer
Task
Task
Submit the code (jar files) and configured dependencies to Executors for further execution

7. What is lazy evaluation in Spark?

Answer: Lazy evaluation in Spark means that Spark does not immediately execute the transformations applied to the data. Instead, it builds a logical execution plan and waits until an action (such as count, collect, or save) is called. This approach allows Spark to optimize the execution plan and improve performance by minimizing data shuffling and combining multiple transformations.

8. What is the basis for splitting stages, jobs, and tasks in Spark?

Answer: In Spark, a job is triggered by an action and consists of multiple stages. Stages are determined by shuffle boundaries, where data needs to be redistributed across the cluster.

Each stage is further divided into tasks, which are the smallest units of work and are executed in parallel on different partitions of the data.

9. **Given the following code, how many actions and stages are there?**

```
val DF_A = read TableA
val DF_B = read TableB
val DF_C = DF_A.join(DF_B)
DF_C.write("tableC")
```

Answer: There is one action (write) and two stages. The first stage involves reading TableA and TableB, and the second stage involves the join operation and writing the result to tableC.

10. **Given the following code, how many actions and stages are there?**

```
val dfA = read TableA
val dfB = dfA.filter(.......)
dfB.write("TableB")
val dfC = dfA.map(.......)
dfC.write("TableC")
```

Answer: There are two actions (write to TableB and write to TableC) and three stages. The first stage involves reading TableA, the second stage involves filtering and writing to TableB, and the third stage involves mapping and writing to TableC.

11. **Given the following code, how many actions and stages are there?**

```
val DF_A = read TableA
val DF_B = read TableB
val DF_C = DF_A.filter(.......)
val DF_D = DF_B.map(........)
DF_C.write("TableC")
DF_D.write("TableD")
```

Answer: There are two actions (write to TableC and write to TableD) and four stages. The first stage involves reading TableA, the second stage involves reading TableB, the third stage involves filtering and writing to TableC, and the fourth stage involves mapping and writing to TableD.

12. **Write a SQL query to print only the names of students who passed all subjects.**

Table: student_marks
Columns: Name, Mark, Subject

Example Data:

Anand, Eng, 50
Anand, Math, 70
Anand, Science, 30
Mahesh, Eng, 90
Mahesh, Math, 92

Mahesh, Science, 78

Output:
Mahesh

SQL Query:

```
SELECT Name
FROM student_marks
GROUP BY Name
HAVING MIN(Mark) >= 40;
```

13. **Write a PySpark code for the above scenario.**

PySpark Code:

```
from pyspark.sql import SparkSession
from pyspark.sql.functions import min

spark = SparkSession.builder.appName("StudentPass").getOrCreate()

# Assuming df is the DataFrame containing student_marks data
df = spark.read.csv("path_to_student_marks.csv", header=True, inferSchema=True)

passed_students = df.groupBy("Name").agg(min("Mark").alias("MinMark")).filter("MinMark >= 40")
passed_students.select("Name").show()
```

14. **In the following code, what will be the first action and second action? What will be the starting point of the second action?**

```
val dfA = read TableA
val dfB = dfA.filter(........)
val dfC = dfA.map(.......)
dfB.write("TableB")
dfC.write("TableC")
```

Answer: The first action is write to TableB, and the second action is write to TableC. The starting point of the second action is after the completion of the first action, which involves reading TableA and filtering it.

15. **Have you heard of the persist command? What is its purpose?**

Answer: Yes, the persist command in Spark is used to store an RDD or DataFrame in memory or on disk across operations. This helps in improving the performance of iterative algorithms and interactive queries by avoiding recomputation of the data.

16. **You have a job with some performance issues. There are some joins that are taking too much time. What will be your approach?**

Answer: To address performance issues with joins, I would:

1. Use broadcast joins for small tables to avoid shuffling large datasets.
2. Ensure proper partitioning of the data to minimize data movement.
3. Optimize the join conditions and filter out unnecessary data before the join.
4. Use caching or persisting intermediate results to avoid recomputation.
5. Tune Spark configurations such as spark.sql.shuffle.partitions and spark.executor.memory.

17. Apart from broadcast, are there other ways to optimize joins?

Answer: Yes, other ways to optimize joins include:

1. Using sort-merge join for large datasets that are already sorted.
2. Using shuffle hash join for smaller datasets.
3. Ensuring proper indexing and partitioning of the data.
4. Using bucketing to colocate related data in the same partitions.

18. Why do you think partitioning will improve performance?

Answer: Partitioning improves performance by:

1. Reducing the amount of data shuffled across the network.
2. Enabling parallel processing of data partitions.
3. Improving data locality by colocating related data.
4. Allowing efficient filtering and pruning of partitions during query execution.

19. Have you worked with pandas?

Answer: Yes, I have worked with pandas for data manipulation and analysis in Python. It provides powerful data structures like DataFrames and Series, which are useful for handling structured data.

20. How are you running your Spark jobs?

Answer: I run Spark jobs using various methods, including:

1. Submitting jobs through the Spark-submit command.
2. Using notebooks such as Jupyter or Databricks.
3. Scheduling jobs with workflow managers like Apache Airflow.
4. Running jobs on cloud platforms like AWS EMR or Azure HDInsight.

21. Do you have anything to ask?

Answer: Yes, I would like to know more about the specific challenges and goals of your data engineering projects. Additionally, I am interested in understanding the team's workflow and the tools and technologies you use.

22. What are the key differences between RDDs, DataFrames, and Datasets in PySpark?

Spark Resilient Distributed Datasets (RDD), DataFrame, and Datasets are key abstractions in Spark that enable us to work with structured data in a distributed computing environment. Even though they are all ways of representing data, they have key differences:

- **RDDs** are low-level APIs that lack a schema and offer control over the data. They are immutable collections of objects

- **DataFrames** are high-level APIs built on top of RDDs optimized for performance but are not safe-type. They organize structured and semi-structured data into named columns.

- **Datasets** combine the benefits of RDDs and DataFrames. They are high-level APIs that provide safe-type abstraction. They support Python and Scala and provide compile-time type checking while being faster than DataFrames.

23. Common Issues in PySpark and How to Resolve Them

1. **Environment Setup Issues**
    - **Problem:** PySpark not installed or environment variables not set correctly.
    - **Solution:**
        - Install PySpark using pip install pyspark.
        - Set JAVA_HOME, HADOOP_HOME, and SPARK_HOME environment variables properly.
        -
2. **Out of Memory Errors**
    - **Problem:** Tasks running out of memory due to large data volumes.
    - **Solution:**
        - Optimize the number of partitions using repartition() or coalesce().
        - Increase executor memory (--executor-memory) and driver memory (--driver-memory) in the configuration.
3. **Skewed Data**
    - **Problem:** Uneven data distribution causing slow performance.
    - **Solution:**
        - Use the salting technique to balance partitions.
        - Use broadcast join for small datasets.
4. **Shuffle Performance Bottlenecks**
    - **Problem:** Excessive shuffling during operations like groupBy or join.
    - **Solution:**
        - Use narrow transformations like map and filter where possible.
        - Adjust spark.sql.shuffle.partitions to reduce shuffle partitions.
5. **Serialization Issues**
    - **Problem:** Incorrect serialization causing errors or slowdowns.
    - **Solution:**
        - Use Kryo serialization by setting spark.serializer to org.apache.spark.serializer.KryoSerializer.
        - Register custom classes if required for better performance.
6. **Schema Mismatch**
    - **Problem:** Input data schema not matching the expected schema.
    - **Solution:**

  - Define schemas explicitly using StructType instead of inferring.
  - Validate schema compatibility before processing.

7. Slow UDF Performance

- **Problem:** Python UDFs slowing down processing.
- **Solution:**
  - Use PySpark’s built-in functions instead of UDFs when possible.
  - Switch to pandas UDFs for better performance.

8. Dependency Conflicts

- **Problem:** Version mismatches between PySpark, Hadoop, or libraries.
- **Solution:**
  - Ensure compatible versions of Spark, Hadoop, and Python are installed.
  - Use virtual environments to manage dependencies.

9. Debugging Challenges

- **Problem:** Limited visibility into distributed jobs.
- **Solution:**
  - Use explain() to analyze query execution plans.
  - Enable Spark UI for monitoring job execution and troubleshooting.

10. File Handling Issues

- **Problem:** Errors while reading/writing data to/from storage.
- **Solution:**
  - Ensure correct file paths and permissions.
  - Use supported file formats like Parquet or ORC for better performance.

11. Inefficient Partitioning

- **Problem:** Too many or too few partitions affecting performance.
- **Solution:**
  - Use df.rdd.getNumPartitions() to check partition count.
  - Adjust partitions using repartition() for better parallelism.

12. Ambiguous Column References

- **Problem:** Errors during operations due to duplicate column names in joins.
- **Solution:**
  - Rename columns before joining using withColumnRenamed().

13. Catalyst Optimizer Limitations

- **Problem:** PySpark's optimizer fails to optimize complex queries.
- **Solution:**
  - Simplify the query logic.
  - Use caching (df.cache()) for repeated computations.

14. Missing Dependencies in Cluster

- **Problem:** Errors due to missing Python or Java libraries on cluster nodes.
- **Solution:**
  - Use --py-files to distribute Python dependencies.
  - Ensure all cluster nodes have the required dependencies installed.

15. Long Execution Time

- **Problem:** Jobs taking too long to execute.
- **Solution:**
  - Profile and optimize transformations.
  - Cache intermediate results to avoid recomputation.

24. Describe how you would optimize a PySpark job that is running slowly. What are the key factors you would look at?

If a PySpark job is running slow, there are several aspects we can improve to optimize its performance:

- Ensuring a proper size and number of data partitions to minimize data shuffling during transformations.
- Using DataFrames instead of RRDs because they already use several Optimization modules to improve the performance of spark workloads.
- Using broadcasting joins and broadcast variables for joining a small dataset with a larger dataset.
- Caching and persisting intermediate DataFrames that are reused.
- Adjusting the number of partitions, executor cores, and instances to effectively use cluster resources.
- Choosing the appropriate file formats to minimize data size.

25. Small file issue

Small File Problem is a common issue in Apache Spark, which can impact the performance of Spark applications. This problem occurs when Spark jobs process a large number of small files, which are typically much smaller in size than the block size of the Hadoop Distributed File System (HDFS).

Causes of the Small File Problem
The Small File Problem in Spark can occur due to the following reasons:

- **Input data partitioning:** When input data is partitioned in a way that creates many small partitions, each partition may contain small files, which can lead to the Small File Problem.
- **Data generation process:** If the data generation process generates many small files instead of larger ones, it can lead to the Small File Problem.
- **Data ingestion process:** If the data ingestion process writes data in a manner that creates many small files, it can lead to the Small File Problem

Resolving the Small File Problem
There are several ways to resolve the Small File Problem in Spark. Here are some of the most common methods:

1. Combine Small Files: You can combine small files into larger files using the "repartition" method in Spark. By combining small files, you can reduce the number of tasks required to process the data, which can improve performance.
2. Increase Block Size: Increasing the block size of HDFS can also help reduce the number of small files. By increasing the block size, you can ensure that files are written in larger blocks, reducing the number of files in the directory.
3. Use Sequence Files: Sequence files are a file format that can store multiple small files in a single file. By using sequence files, you can reduce the number of files in a directory and improve performance.
4. Use Hadoop Archive Files: Hadoop archive files (HAR) are another file format that can store multiple small files in a single file. By using HAR files, you can reduce the number of files in a directory and improve performance.

26. Assume you have a dataset of 500 GB that needs to be processed on a Spark cluster. The cluster has 10 nodes, each with 64 GB of memory and 16 cores. How would you allocate resources for your Spark job?

Resource allocation in Spark depends on multiple factors such as the size of the dataset, the complexity of computations, and the configuration of the cluster. Here are some guidelines for different scenarios:

Scenario 1: General Guidelines

- **Number of Executors:** Reserve some resources for the Operating System and Hadoop daemons. For example, reserve 1 core and 1 GB per node. This leaves 15 cores and 63 GB per node for Spark. To avoid network I/O during shuffles, it's best to have as many executors as nodes. So, you could have 10 executors.
- **Memory per Executor:** Allocate the total memory available per node, i.e., 63 GB.
- **Cores per Executor:** Allocate the total cores available per node, i.e., 15 cores. For better concurrency, assign around 5 cores per executor, leading to around 3 executors per node.
- **Driver Memory:** The driver program can be run on a separate node, or if on the same cluster, allocate it 1–2 cores and about 5–10% of the total memory.

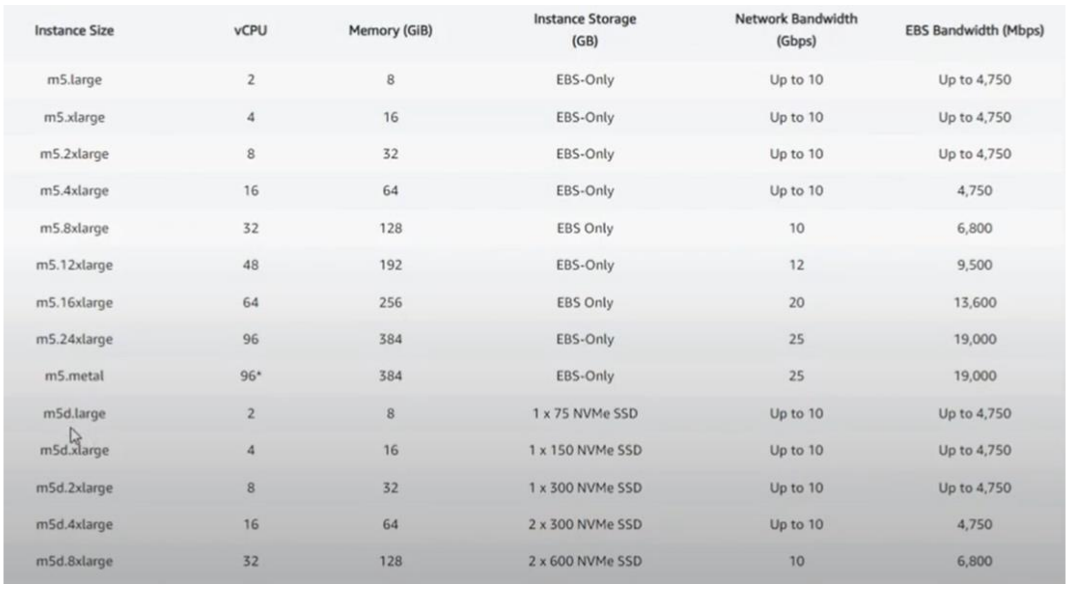
Scenario 2: Processing 1 TB of Data with 5 Nodes (8 Cores, 32 GB RAM Each)

- **Number of Executors:** Reserve 1 core for Hadoop and OS daemons, leaving 7 cores per node. With 5 nodes, you have a total of 35 available cores. Allocating around 5 cores per executor, you would have 7 executors.
- **Memory per Executor:** Reserve 1 GB for the OS and Hadoop daemons, leaving 31 GB. Allocate around 27 GB to the executor, leaving some off-heap memory.
- **Cores per Executor:** Keep it at 5 cores per executor.
- **Driver Memory:** Assign around 3–4 GB of memory to the driver, and run it on a separate node if possible to avoid resource competition.
- **Memory and Core Allocation for Hadoop Daemons:** 1 GB and 1 core respectively.

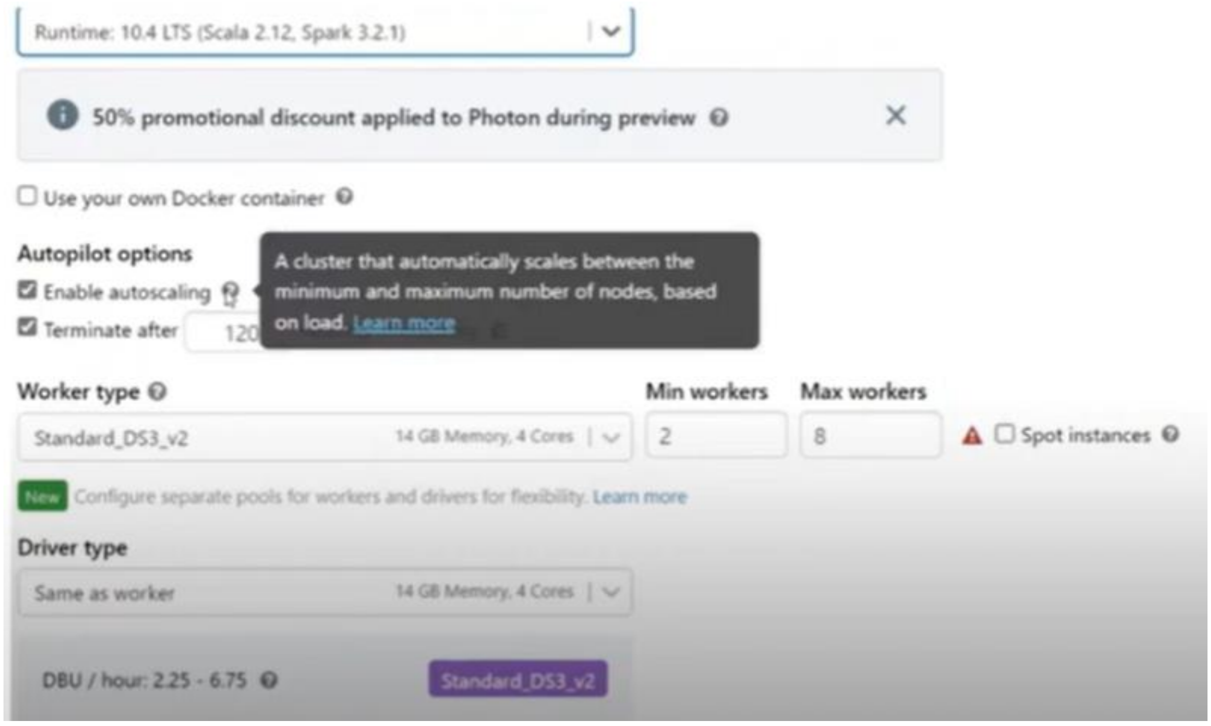
Scenario 3: Processing 5 TB of Data with 50 Nodes (16 Cores, 128 GB RAM Each)

- **Number of Executors:** After reserving resources for the OS and daemons, you are left with 15 cores and 127 GB per node. Run 15 executors per node, for a total of 750 executors.
- **Memory per Executor:** With 127 GB available on each node, allocate approximately 8 GB per executor, allowing for some off-heap usage.
- **Cores per Executor:** Allocate 1 core per executor to maximize parallelism and reduce task scheduling overhead.
- **Driver Memory:** Run the driver on a separate node if possible, and allocate about 10–20 GB, as it needs to collect task states from a large number of executors.
- **Data Serialization:** Consider using Kryo serialization for more efficient serialization of data.

27. Worker Node Size:



Instance Size	vCPU	Memory (GiB)	Instance Storage (GB)	Network Bandwidth (Gbps)	EBS Bandwidth (Mbps)
m5.large	2	8	EBS-Only	Up to 10	Up to 4,750
m5.xlarge	4	16	EBS-Only	Up to 10	Up to 4,750
m5.2xlarge	8	32	EBS-Only	Up to 10	Up to 4,750
m5.4xlarge	16	64	EBS-Only	Up to 10	4,750
m5.8xlarge	32	128	EBS Only	10	6,800
m5.12xlarge	48	192	EBS-Only	12	9,500
m5.16xlarge	64	256	EBS Only	20	13,600
m5.24xlarge	96	384	EBS-Only	25	19,000
m5.metal	96*	384	EBS-Only	25	19,000
m5d.large	2	8	1 x 75 NVMe SSD	Up to 10	Up to 4,750
m5d.xlarge	4	16	1 x 150 NVMe SSD	Up to 10	Up to 4,750
m5d.2xlarge	8	32	1 x 300 NVMe SSD	Up to 10	Up to 4,750
m5d.4xlarge	16	64	2 x 300 NVMe SSD	Up to 10	4,750
m5d.8xlarge	32	128	2 x 600 NVMe SSD	10	6,800



Runtime: 10.4 LTS (Scala 2.12, Spark 3.2.1)

50% promotional discount applied to Photon during preview

☐ Use your own Docker container

Autopilot options

☑ Enable autoscaling

☑ Terminate after 120

A cluster that automatically scales between the minimum and maximum number of nodes, based on load. Learn more

Worker type		Min workers	Max workers	
Standard_DS3_v2	14 GB Memory, 4 Cores	2	8	☐ Spot instances

New Configure separate pools for workers and drivers for flexibility. Learn more

Driver type

Same as worker 14 GB Memory, 4 Cores

DBU / hour: 2.25 - 6.75 Standard_DS3_v2

28. How to Debugging and Fixing Out of Memory Errors

If a Spark job is running out of memory with the error: "java.lang.OutOfMemoryError: Java heap space", here are some steps to debug and fix the issue:

- **Increase Executor Memory:** Increase the executor memory with the spark.executor.memory property.
- **Increase Driver Memory:** If the driver is running out of memory, increase it with the spark.driver.memory property.

- **Memory Management:** Analyze how memory is being used in your Spark job. If caching is excessive, consider reducing it or using the MEMORY_AND_DISK storage level to spill to disk when necessary.
- **Data Serialization:** Use Kryo serialization, which is more memory-efficient than Java serialization.
- **Partitioning:** If some tasks handle significantly more data than others, you might have a data skew problem. Re-partitioning the data might help.

29. How to Diagnosing and Improving Spark Application Performance?

If your Spark application is running slower than expected, here are some steps to diagnose and improve performance:

- **Check Resource Utilization:** Use Spark's web UI or other monitoring tools to check CPU and memory utilization. Low CPU usage could indicate an I/O or network bottleneck, while high garbage collection times could indicate memory issues.
- **Data Skew:** If some tasks take much longer than others, you might have a data skew problem. Consider repartitioning your data.
- **Serialization:** If a lot of time is spent on serialization and deserialization, switch to Kryo serialization, which is more efficient.
- **Tuning Parallelism:** Adjust the level of parallelism. Too few partitions can lead to less concurrency, while too many can lead to excessive overhead. A rule of thumb is to have 2–3 tasks per CPU core in your cluster.
- **Caching:** If your application reuses intermediate RDDs or DataFrames, use caching to avoid recomputation.
- **Tune Spark Configuration:** Depending on the characteristics of your application and dataset, you may need to tune various Spark configurations. For example, increase spark.driver.memory, spark.executor.memory, or spark.network.timeout, or decrease spark.memory.fraction.

30. What is coalesce in Spark?

Answer: coalesce is a transformation in Spark that reduces the number of partitions in a DataFrame or RDD. It is often used to optimize the performance of a job by reducing the number of partitions to a specified number, which can be useful when you have a large number of small partitions. Unlike repartition, coalesce avoids a full shuffle of the data.

31. What is repartition in Spark?

Answer: repartition is a transformation in Spark that reshuffles the data in a DataFrame or RDD to increase or decrease the number of partitions. This operation involves a full shuffle of the data across the cluster, which can be useful for balancing the data distribution or increasing parallelism.

32. What is the difference between cache() and persist() in Spark?

Answer: Both cache() and persist() are used to store DataFrames or RDDs in memory to speed up subsequent actions. The difference is:

- cache(): By default, it stores the data in memory only.
- persist(): Allows you to specify different storage levels (e.g., memory, disk, or a combination) using the StorageLevel class.
-

33. **What are actions and transformations in Spark?**

Answer:

- **Transformations:** These are operations that create a new DataFrame or RDD from an existing one. They are lazily evaluated, meaning they are not executed until an action is called. Examples include map(), filter(), flatMap(), groupBy(), and join().
- **Actions:** These are operations that trigger the execution of transformations and return a result to the driver program or write data to an external storage system. Examples include collect(), count(), saveAsTextFile(), and reduce().

34. What is the difference between map and flatMap in Spark?

Answer:

- map: Applies a function to each element of the DataFrame or RDD and returns a new DataFrame or RDD with the results. The number of elements remains the same.
- flatMap: Similar to map, but the function applied can return a list of elements, which are then flattened into a single DataFrame or RDD. This can change the number of elements.

35. When will shuffling happen in Spark?

Answer: Shuffling occurs when data is redistributed across the cluster, which can happen during operations like groupByKey(), reduceByKey(), join(), distinct(), and repartition(). Shuffling involves moving data between partitions and can be an expensive operation in terms of performance.

36. How do you read a Spark file with a delimiter | or \t in a DataFrame?

Answer: You can specify the delimiter using the option method when reading the file. Here is an example:

```
from pyspark.sql import SparkSession

spark = SparkSession.builder.appName("ReadFile").getOrCreate()

# Reading a file with delimiter '|'
df_pipe = spark.read.option("delimiter", "|").csv("path_to_file", header=True, inferSchema=True)

# Reading a file with delimiter '\t'
df_tab = spark.read.option("delimiter", "\t").csv("path_to_file", header=True, inferSchema=True)
```

37. **Explain a scenario where you apply an optimization technique in Spark.**

Answer: One common optimization technique is using broadcast joins to handle skewed data. For example, if you have a large DataFrame df_large and a small

DataFrame df_small, you can use a broadcast join to avoid shuffling the large DataFrame:

```
from pyspark.sql.functions import broadcast

df_large = spark.read.csv("path_to_large_file", header=True, inferSchema=True)
df_small = spark.read.csv("path_to_small_file", header=True, inferSchema=True)

# Using broadcast join to optimize performance
df_joined = df_large.join(broadcast(df_small), df_large["key"] == df_small["key"])
```

This technique helps in reducing the shuffle overhead and improves the performance of the join operation.

38. Find the Duplicates from a Table

SQL Query:

```
SELECT column_name, COUNT(*)
FROM your_table
GROUP BY column_name
HAVING COUNT(*) > 1;
```

PySpark DataFrame Code:

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

spark = SparkSession.builder.appName("FindDuplicates").getOrCreate()

df = spark.read.csv("path_to_file", header=True, inferSchema=True)

duplicates = df.groupBy("column_name").count().filter(col("count") > 1)
duplicates.show()
```

39. Find the 2nd Highest Salary from a Table

SQL Query:

```
SELECT salary
FROM (
  SELECT salary, ROW_NUMBER() OVER (ORDER BY salary DESC) AS row_num
  FROM your_table
)
WHERE row_num = 2;
```

PySpark DataFrame Code:

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

spark = SparkSession.builder.appName("SecondHighestSalary").getOrCreate()

df = spark.read.csv("path_to_file", header=True, inferSchema=True)

windowSpec = Window.orderBy(df["salary"].desc())
```

```
df_with_row_num = df.withColumn("row_num", row_number().over(windowSpec))
second_highest_salary = df_with_row_num.filter(df_with_row_num["row_num"] ==
2).select("salary")
second_highest_salary.show()
```

40. Lag Question Regarding ID Column: How to Add Previous ID to Next Same Row

SQL Query:

```
SELECT id, LAG(id) OVER (ORDER BY id) AS previous_id
FROM your_table;
```

PySpark DataFrame Code:

```
from pyspark.sql import SparkSession
from pyspark.sql.window import Window
from pyspark.sql.functions import lag

spark = SparkSession.builder.appName("LagFunction").getOrCreate()

df = spark.read.csv("path_to_file", header=True, inferSchema=True)

windowSpec = Window.orderBy("id")
df_with_lag = df.withColumn("previous_id", lag("id").over(windowSpec))
df_with_lag.show()
```

41. Left Join in PySpark

Explanation: A left join returns all the rows from the left DataFrame and the matching rows from the right DataFrame. If there is no match, the result will contain null values for columns from the right DataFrame.

Use: Left joins are useful when you want to keep all records from the left DataFrame and include corresponding records from the right DataFrame if they exist.

Example:

```
from pyspark.sql import SparkSession

spark = SparkSession.builder.appName("LeftJoin").getOrCreate()

df_left = spark.read.csv("path_to_left_file", header=True, inferSchema=True)
df_right = spark.read.csv("path_to_right_file", header=True, inferSchema=True)

df_left_join = df_left.join(df_right, df_left["key"] == df_right["key"], "left")
df_left_join.show()
```

42. Right Join in PySpark

Explanation: A right join returns all the rows from the right DataFrame and the matching rows from the left DataFrame. If there is no match, the result will contain null values for columns from the left DataFrame.

Use: Right joins are useful when you want to keep all records from the right DataFrame and include corresponding records from the left DataFrame if they exist.

Example:

```
from pyspark.sql import SparkSession

spark = SparkSession.builder.appName("RightJoin").getOrCreate()

df_left = spark.read.csv("path_to_left_file", header=True, inferSchema=True)
df_right = spark.read.csv("path_to_right_file", header=True, inferSchema=True)

df_right_join = df_left.join(df_right, df_left["key"] == df_right["key"], "right")
df_right_join.show()
```

43. Inner Join in PySpark

Explanation: An inner join returns only the rows that have matching values in both DataFrames. If there is no match, the row is excluded from the result.

Use: Inner joins are useful when you want to retrieve only the records that have corresponding matches in both DataFrames.

Example:

```
from pyspark.sql import SparkSession

spark = SparkSession.builder.appName("InnerJoin").getOrCreate()

df_left = spark.read.csv("path_to_left_file", header=True, inferSchema=True)
df_right = spark.read.csv("path_to_right_file", header=True, inferSchema=True)

df_inner_join = df_left.join(df_right, df_left["key"] == df_right["key"], "inner")
df_inner_join.show()
```

44. Full Outer Join in PySpark

Explanation: A full outer join returns all the rows when there is a match in either the left or right DataFrame. If there is no match, the result will contain null values for columns from the DataFrame that does not have a match.

Use: Full outer joins are useful when you want to retrieve all records from both DataFrames, regardless of whether there is a match.

Example:

```
from pyspark.sql import SparkSession

spark = SparkSession.builder.appName("FullOuterJoin").getOrCreate()

df_left = spark.read.csv("path_to_left_file", header=True, inferSchema=True)
df_right = spark.read.csv("path_to_right_file", header=True, inferSchema=True)

df_full_outer_join = df_left.join(df_right, df_left["key"] == df_right["key"], "outer")
df_full_outer_join.show()
```

45. Left Anti Join in PySpark

Explanation: A left anti join returns only the rows from the left DataFrame that do not have a match in the right DataFrame.

Use: Left anti joins are useful when you want to find records in the left DataFrame that do not have corresponding matches in the right DataFrame.

Example:

```
from pyspark.sql import SparkSession

spark = SparkSession.builder.appName("LeftAntiJoin").getOrCreate()

df_left = spark.read.csv("path_to_left_file", header=True, inferSchema=True)
df_right = spark.read.csv("path_to_right_file", header=True, inferSchema=True)

df_left_anti = df_left.join(df_right, df_left["key"] == df_right["key"], "left_anti")
df_left_anti.show()
```

46. Left Semi Join in PySpark

Explanation: A left semi join returns only the rows from the left DataFrame that have a match in the right DataFrame. It is similar to an inner join, but it returns only columns from the left DataFrame.

Use: Left semi joins are useful when you want to filter the left DataFrame to include only rows that have corresponding matches in the right DataFrame.

Example:

```
from pyspark.sql import SparkSession

spark = SparkSession.builder.appName("LeftSemiJoin").getOrCreate()

df_left = spark.read.csv("path_to_left_file", header=True, inferSchema=True)
df_right = spark.read.csv("path_to_right_file", header=True, inferSchema=True)

df_left_semi = df_left.join(df_right, df_left["key"] == df_right["key"], "left_semi")
df_left_semi.show()
```

47. Cross Join in PySpark

Explanation: A cross join returns the Cartesian product of the two DataFrames, meaning it returns all possible combinations of rows from the left and right DataFrames.

Use: Cross joins are useful when you need to generate all possible combinations of rows from two DataFrames. However, they can be very expensive in terms of computation and memory, so use them with caution.

Example:

```
from pyspark.sql import SparkSession

spark = SparkSession.builder.appName("CrossJoin").getOrCreate()
```

```
df_left = spark.read.csv("path_to_left_file", header=True, inferSchema=True)
df_right = spark.read.csv("path_to_right_file", header=True, inferSchema=True)

df_cross_join = df_left.crossJoin(df_right)
df_cross_join.show()
```

48. Anti Join in PySpark

Explanation: An anti join returns only the rows from the left DataFrame that do not have a match in the right DataFrame. It is similar to a left anti join but can be used in different contexts.

Use: Anti joins are useful when you want to exclude records from the left DataFrame that have corresponding matches in the right DataFrame.

Example:

```
from pyspark.sql import SparkSession

spark = SparkSession.builder.appName("AntiJoin").getOrCreate()

df_left = spark.read.csv("path_to_left_file", header=True, inferSchema=True)
df_right = spark.read.csv("path_to_right_file", header=True, inferSchema=True)

df_anti = df_left.join(df_right, df_left["key"] == df_right["key"], "anti")
df_anti.show()
```

49. **Write python on code based on below explanation.**

df1 = productcode, charid, productid, items , sales_units
df2 = productcode, charid, productid

from above schema write program to
1) get the sum of total items grouped on charid
2) get the sumof sales grouped on sales_units

Answer:

```
from pyspark.sql import SparkSession
from pyspark.sql.functions import sum

# Initialize Spark session
spark = SparkSession.builder.appName("AggregationExample").getOrCreate()

# Sample data for df1
data1 = [
    ("P1", "C1", "PR1", 10, 100),
    ("P2", "C1", "PR2", 20, 200),
    ("P3", "C2", "PR3", 30, 300),
    ("P4", "C2", "PR4", 40, 400)
]

# Sample data for df2
data2 = [
    ("P1", "C1", "PR1"),
    ("P2", "C1", "PR2"),
    ("P3", "C2", "PR3"),
    ("P4", "C2", "PR4")
]

# Define schema for df1
schema1 = ["productcode", "charid", "productid", "items", "sales_units"]

# Define schema for df2
schema2 = ["productcode", "charid", "productid"]

# Create DataFrames
df1 = spark.createDataFrame(data1, schema1)
df2 = spark.createDataFrame(data2, schema2)

# 1. Get the sum of total items grouped by charid
sum_items = df1.groupBy("charid").agg(sum("items").alias("total_items"))
sum_items.show()

# 2. Get the sum of sales grouped by sales_units
sum_sales = df1.groupBy("sales_units").agg(sum("sales_units").alias("total_sales"))
sum_sales.show()
```

Explanation:

1. **Initialize Spark session:** This sets up the Spark environment.
2. **Sample data for df1 and df2:** These are the sample data based on the given schema.
3. **Define schema for df1 and df2:** These are the column names for the DataFrames.
4. **Create DataFrames:** The sample data is converted into Spark DataFrames.
5. **Get the sum of total items grouped by charid:** The groupBy method is used to group the data by charid, and the agg method is used to calculate the sum of items.
6. **Get the sum of sales grouped by sales_units:** The groupBy method is used to group the data by sales_units, and the agg method is used to calculate the sum of sales_units.

50. How to handle Null value in PySpark.

Dropping Null Values:

You can drop rows or columns that contain null values using the dropna() method.

```
df.dropna().show()  # Drops rows with any null values
df.dropna(how='all').show()  # Drops rows where all values are null
df.dropna(subset=['column1', 'column2']).show()  # Drops rows where specified columns have null values
```

Filling Null Values:

You can fill null values with a specific value using the fillna() method.

```
df.fillna(0).show()  # Fills all null values with 0
df.fillna({'column1': 0, 'column2': 'missing'}).show()  # Fills null values in specified columns with different values
```

Replacing Null Values:

```
df.na.replace('null_value', 'replacement_value').show()
```

Using SQL Functions:

```
from pyspark.sql.functions import coalesce
df.withColumn('new_column', coalesce(df['column1'], df['column2'])).show()
```

51. How Handle Good and Bad Records using pySpark.

52. Introduce about your technology and experience.
53. Explain current project in detail with your role on daily basis.
54. How many services you used in AWS name it

55. What is functionality of glue and lambda in AWS.
56. What kinds of storage you used in your current project and why.
57. How did you submit spark job in your current project.
58. How deployment happen in your project.
59. How did you work with agile methodology in your project and what all ceremony we have. Tool, you used for agile.
60. How comfortable are you in git and what command did you used.
61. How frequently you commit code.
62. What is docker.
63. Diff b/w method and function.
64. If the task assigned to you is not completed on given time, how would you handle it.
65. If your peer/colleague is not able to complete with their task what you will do.
66. What is spark.
67. If spark job fails, how would you start looking on issues.
68. What all challenges you faced and solution/optimization you did in for that.
69. Diff b/w RDD and data frame.
70. How to create dataset.
71. What is anonymous functions.

72. What is notebook in databrics and how you have deployed notebook in you project.

73. What orchestration tolls you used in databrics.

74. How comfortable you are on delta lake, data lake and delta table.

75. if we use narrow transformation how many stages will created.

76. What is narrow and wide transformation with example.

77. What all are window functions and how frequently you used window function and name it.

78. What is method overwrite and method overload explain in details.

79. What is table and views in sql.

80. Scenario based questions.

81. Count the occurrence of each element from a list. Write a program that a given list count the occurrence of each element
82. What is repartition?
83. How you do repartition by a column?
84. How spark handles when the application runs?
85. If we want to reduce the time spark takes for a job, what could be the procedure?
86. How you can debug a slow running spark?
87. How do you handle uneven distribution in the partitions?
88. What is broadcast?
89. If two tables are large how spark handles the join operation?

Coding Question:::

ProductId	Category	Units	SalesValue
1001	1	10	1000
1002	1	10	2000
1003	1	10	3000
1004	2	10	4000
1005	2	10	2000
1006	3	5	800
1007	3	5	600
1008	3	5	500
1009	4	3	400
1010	4	2	200
1010	6	2	200
1001	1	5	500
1001	1	5	500
1001	1	5	500

1003	1	10	3000

Category	Description
1	Cat-1
2	Cat-2
3	Cat-3
4	Cat-4
5	Cat-5

ProductId	Category	Units	SalesValue	CategoryTotalSales
1001	1	10	1000	10500
1002	1	10	2000	10500
1003	1	10	3000	10500
1004	2	10	4000	6000
1005	2	10	2000	6000
1006	3	5	800	1900
1007	3	5	600	1900
1008	3	5	500	1900
1009	4	3	400	600
1010	4	2	200	600
1010	6	2	200	200
1001	1	5	500	10500
1001	1	5	500	10500
1001	1	5	500	10500
1003	1	10	3000	10500

1. Join two tables Need to have all categories in second table and should have 0 as total sales value if it is not present in the first table.
2. Create the last table.
3. Following up get the category having second highest total sales.

(i) Narrow vs Wide Transformation
(ii) How Airflow dag works in ~~A~~ Spark?
(iii) Persist ?
(iv) why partitioning will improve and why
(v) RDD vs Dataframe vs datasets
(vi) Common issues in pyspark and how to resolve
(v) How to allocate resources for 500 GB data process?
(vi) Colesce and Repartition
(vii) Cache vs ~~B.~~persist~~ent~~
~~&~~ ~~A~~ Shuffling, when it will happen?
ix) inner join, in pyspark,
will give a table and write codes,
Product & Category
↳ ID ↳ Cat, Cattype

Join get sales value, New column add
[illegible]

Airflow → How DAG works ? What is it?