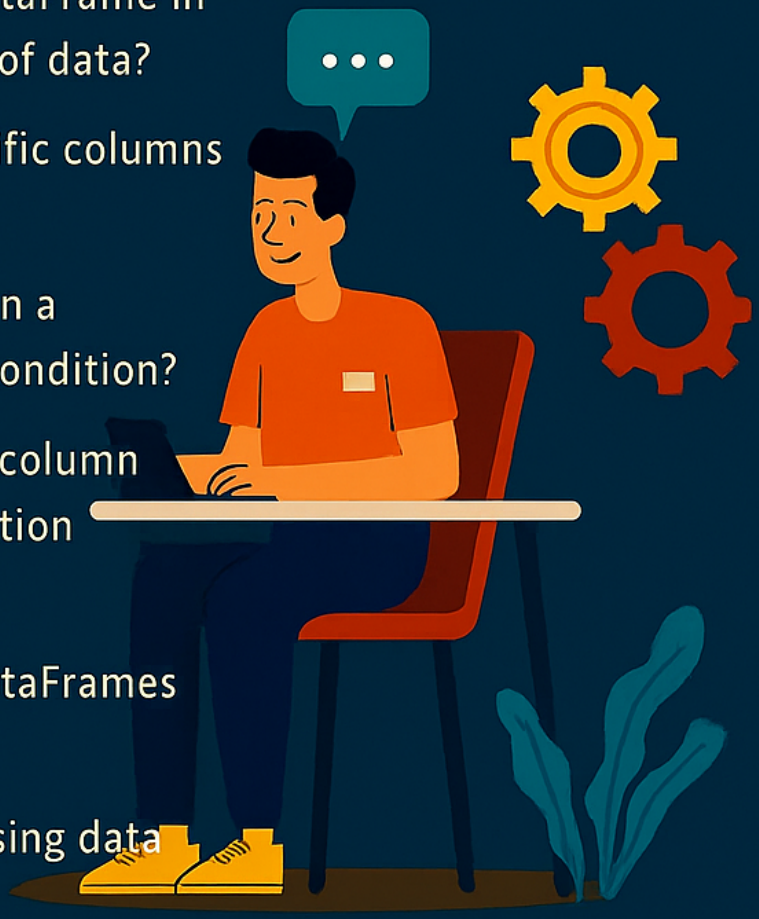
10 PYSPARK DATAFRAME INTERVIEW QUESTIONS AND CLIENT AND CLUSTER MODE

- How do you create a DataFrame in Spark from a collection of data?
- How do you select specific columns from a DataFrame?
- How do you filter rows in a DataFrame based on a condition?
- How do you group by a column and perform an aggregation in Spark DataFrame?
- How do you join two DataFrames in Spark?
- How do you handle missing data in Spark DataFrame?
- How do you sort a DataFrame by a specific column?
- How do you add a new column to a DataFrame?
- How do you remove duplicate rows from a DataFrame?



ARVIND KUMAR

10 Pyspark Dataframe interview questions with solution

Here are 10 critical interview questions on Spark DataFrame operations along with their solutions:

Question 1: How do you create a DataFrame in Spark from a collection of data?

Solution:

```
from pyspark.sql import SparkSession

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

# Sample data
data = [("John", 25), ("Doe", 30), ("Jane", 28)]

# Create DataFrame
columns = ["name", "age"]
df = spark.createDataFrame(data, columns)

# Show DataFrame
df.show()

# Stop Spark session
spark.stop()
```

Question 2: How do you select specific columns from a DataFrame?

Solution:

```
# Select specific columns
selected_df = df.select("name", "age")

# Show DataFrame
selected_df.show()
```

Question 3: How do you filter rows in a DataFrame based on a condition?

Solution:

```
# Filter rows where age is greater than 25
filtered_df = df.filter(df["age"] > 25)

# Show DataFrame
filtered_df.show()
```

Question 4: How do you group by a column and perform an aggregation in Spark DataFrame?

Solution:

```
# Sample data
data = [("John", "HR", 3000), ("Doe", "HR", 4000),("Jane",
"IT", 5000), ("Mary", "IT", 6000)]

# Create DataFrame
columns = ["name", "department", "salary"]
df = spark.createDataFrame(data, columns)

# Group by department and calculate average salary
avg_salary_df = df.groupBy("department").avg("salary")
```

```
# Show the result
avg_salary_df.show()
```

Question 5: How do you join two DataFrames in Spark?

Solution:

```
# Sample data
data1 = [("John", 1), ("Doe", 2), ("Jane", 3)]
data2 = [(1, "HR"), (2, "IT"), (3, "Finance")]

# Create DataFrames
columns1 = ["name", "dept_id"]
columns2 = ["dept_id", "department"]

df1 = spark.createDataFrame(data1, columns1)
df2 = spark.createDataFrame(data2, columns2)

# Join DataFrames on dept_id
joined_df = df1.join(df2, "dept_id")

# Show the result
joined_df.show()
```

Question 6: How do you handle missing data in Spark DataFrame?

Solution:

```
# Sample data
data = [("John", None), ("Doe", 25), ("Jane", None), ("Mary", 30)]

# Create DataFrame
columns = ["name", "age"]
df = spark.createDataFrame(data, columns)

# Fill missing values with a default value
df_filled = df.fillna({'age': 0})

# Show the result
df_filled.show()
```

Question 7: How do you apply a custom function to a DataFrame column using UDF?

Solution:

```
from pyspark.sql.functions
import udf from pyspark.sql.types
import StringType


# Define UDF to convert department to uppercase
def convert_uppercase(department):
       return department.upper()


# Register UDF
convert_uppercase_udf = udf(convert_uppercase, StringType())

# Apply UDF to DataFrame
df_transformed = df.withColumn("department_upper", convert_uppercase_udf(df["department"]))

# Show the result
df_transformed.show()
```

Question 8: How do you sort a DataFrame by a specific column?

Solution:

```
# Sort DataFrame by age
sorted_df = df.orderBy("age")


# Show the result
sorted_df.show()
```

Question 9: How do you add a new column to a DataFrame?

Solution:

```
# Add a new column with a constant value
df_with_new_column = df.withColumn("new_column", df["age"] * 2)

# Show the result
df_with_new_column.show()
```

Question 10: How do you remove duplicate rows from a DataFrame?

Solution:

```
# Sample data with duplicates
data = [("John", 25), ("Doe", 30), ("Jane", 28), ("John", 25)]

# Create DataFrame
columns = ["name", "age"]
df = spark.createDataFrame(data, columns)

# Remove duplicate rows
df_deduplicated = df.dropDuplicates()

# Show the result
df_deduplicated.show()
```

These questions and solutions cover fundamental and advanced operations with Spark DataFrames, which are essential for data processing and analysis using Spark.

PySpark Client Mode and Cluster Mode

Apache Spark can run in multiple deployment modes, including client and cluster modes, which determine where the Spark driver program runs and how tasks are scheduled across the cluster. Understanding the differences between these modes is essential for optimizing Spark job performance and resource utilization.

1. PySpark Client Mode

Client mode is a deployment mode where the Spark driver runs on the machine where the `spark-submit` command is executed. The driver program communicates with the cluster's executors to schedule and execute tasks.

Key Characteristics of Client Mode:

- **Driver Location**: Runs on the machine where the user launches the application.
- **Best for Interactive Use**: Ideal for development, debugging, and interactive sessions like using notebooks (e.g., Jupyter) where you want immediate feedback.
- **Network Dependency**: The driver needs to maintain a constant connection with the executors. If the network connection between the client machine and the cluster is unstable, the job can fail.
- **Resource Utilization**: The client machine's resources (CPU, memory) are used for the driver, so a powerful client machine is beneficial.

Code Implementation for Client Mode:

To run a PySpark application in client mode, you would use the `spark-submit` command with `--deploy-mode client`. Here's an example:

```
spark-submit \
  --master yarn \
  --deploy-mode client \
  --num-executors 3 \
  --executor-cores 2 \
  --executor-memory 4G \
  --driver-memory 2G \
  my_pyspark_script.py
```

Explanation:

- **`--master yarn`**: Specifies YARN as the cluster manager.
- **`--deploy-mode client`**: Runs the driver on the client machine where the command is executed.
- **`--num-executors`, `--executor-cores`, `--executor-memory`**: Configures the number of executors, CPU cores per executor, and memory allocation per executor.
- **`--driver-memory`**: Allocates memory for the driver program on the client machine.
- **`my_pyspark_script.py`**: The PySpark script that contains your Spark application code.

PySpark Script Example:

```
from pyspark.sql import SparkSession

# Initialize SparkSession
spark = SparkSession.builder \
    .appName("ClientModeExample") \
    .getOrCreate()

# Sample DataFrame creation
data = [("John", 30), ("Doe", 25), ("Alice", 29)]
columns = ["Name", "Age"]
df = spark.createDataFrame(data, columns)

# Perform operations
```

```
df.show()
df.groupBy("Age").count().show()

# Stop SparkSession
spark.stop()
```

2. PySpark Cluster Mode

Cluster mode is a deployment mode where the Spark driver runs inside the cluster, typically on one of the worker nodes, and not on the client machine. This mode is more suitable for production jobs that require high availability and reliability.

Key Characteristics of Cluster Mode:

- **Driver Location**: Runs on one of the cluster's worker nodes.
- **Best for Production**: Suitable for production environments where long-running jobs need stability and don't require interactive sessions.
- **Less Network Dependency**: Since the driver is located within the cluster, it has more stable connections with executors, reducing the risk of job failures due to network issues.
- **Resource Management**: Utilizes cluster resources for the driver, freeing up client resources and often providing more powerful hardware for the driver process.

Code Implementation for Cluster Mode:

To run a PySpark application in cluster mode, you use `spark-submit` with `--deploy-mode cluster`. Here's an example:

```
spark-submit \
  --master yarn \
  --deploy-mode cluster \
  --num-executors 5 \
  --executor-cores 4 \
  --executor-memory 8G \
  --driver-memory 4G \
  --conf spark.yarn.submit.waitAppCompletion=false \
  my_pyspark_script.py
```

Explanation:

- **`--master yarn`**: Specifies YARN as the cluster manager.
- **`--deploy-mode cluster`**: Runs the driver on a worker node within the cluster.
- **`--num-executors`, `--executor-cores`, `--executor-memory`**: Configures the number of executors, CPU cores per executor, and memory allocation per executor.
- **`--driver-memory`**: Allocates memory for the driver program within the cluster.
- **`--conf spark.yarn.submit.waitAppCompletion=false`**: Submits the application and returns immediately without waiting for job completion. This is useful for running jobs asynchronously in a production environment.
- **`my_pyspark_script.py`**: The PySpark script that contains your Spark application code.

PySpark Script Example:

```
from pyspark.sql import SparkSession

# Initialize SparkSession
spark = SparkSession.builder \
    .appName("ClusterModeExample") \
    .getOrCreate()

# Load data from HDFS
df = spark.read.csv("hdfs:///path/to/input.csv", header=True, inferSchema=True)
```

```
# Perform operations
result_df = df.filter(df['age'] > 30).groupBy("city").count()

# Save the result back to HDFS
result_df.write.csv("hdfs:///path/to/output.csv")

# Stop SparkSession
spark.stop()
```

Choosing Between Client Mode and Cluster Mode

- **Use Client Mode**:
  - For interactive analysis or debugging using notebooks.
  - When you need immediate feedback and are running jobs from your local machine.
  - For smaller workloads where the driver's resource needs are minimal.
- **Use Cluster Mode**:
  - For production jobs that require high reliability and scalability.
  - When running long-running batch jobs or when the driver needs significant resources.
  - When you want to avoid network instability affecting the driver's connection to the executors.

Conclusion

Understanding the differences between client mode and cluster mode in PySpark is crucial for effectively managing resources and optimizing job performance. Client mode is great for development and debugging, while cluster mode is ideal for production environments where stability and resource management are critical. By leveraging these modes appropriately, you can ensure your Spark jobs run efficiently and reliably.