

+1 530 264 8480 (tel:+15302648480)

✉ contact@k21academy.com (mailto:contact@k21academy.com)


☎ Whatsapp (<http://k21academy.com/whatsapp>)

Cloud

Top 60+ PySpark Interview Questions and Answers



July 6, 2024 by [Aniket Gawandar](https://k21academy.com/author/aniketk21academy-com/) (https://k21academy.com/author/aniketk21academy-com/)

 22328 views

Preparing for a **PySpark or data engineering job interview** can be challenging, especially when facing **complex PySpark interview questions**. Whether you're an **aspiring data analyst, data scientist, or data engineer**, it's important to be well-prepared.

To help you out, we've selected the **top 70 most important PySpark interview questions and answers**. This guide will arm you with the **expertise and assurance** required to excel in your interview and land your **dream job in data management**.

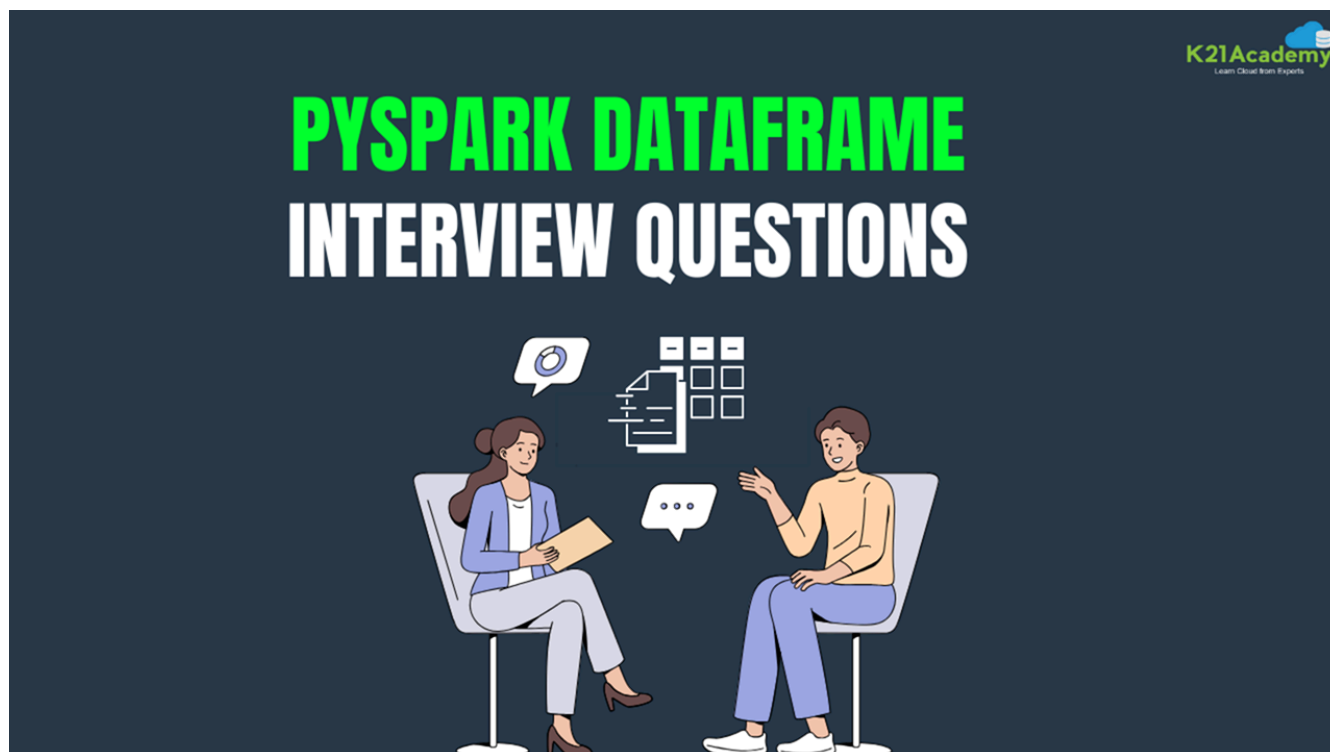
1. [Introduction to PySpark](#)
2. [PySpark DataFrame Interview Questions](#)
3. [PySpark Coding Interview Questions](#)
4. [PySpark Interview Questions for Experienced Data Engineers](#)
5. [Interview Questions on PySpark Data Science](#)
6. [Advanced PySpark Interview Questions and Answers](#)
7. [PySpark Scenario-Based Interview Questions for Experienced Professionals](#)
8. [Capgemini PySpark Interview Questions](#)
9. [Conclusion](#)
10. [Frequently Asked Questions](#)

Introduction to PySpark

PySpark is an interface for Apache Spark in Python. It allows you to write Spark applications using Python APIs, and the PySpark shell provides an interactive interface to work with data in Spark. As data engineering and big data analytics grow, mastering PySpark is essential for professionals in these fields. Let's dive into the top 70 PySpark interview questions and answers to equip you for your next interview better.



PySpark DataFrame Interview Questions [^]_^



Q1) What's the difference between an RDD, a DataFrame, and a DataSet?

RDD (Resilient Distributed Dataset):

- It is the **fundamental data structure** of **Spark**. **RDDs** are the building blocks for DataFrames and Datasets.
- **RDDs** can be efficiently cached if a similar set of data needs to be computed multiple times.
- Useful for performing low-level transformations, operations, and control on a dataset.
- Primarily used for manipulating data with **functional programming constructs** rather than domain-specific expressions.

DataFrame:

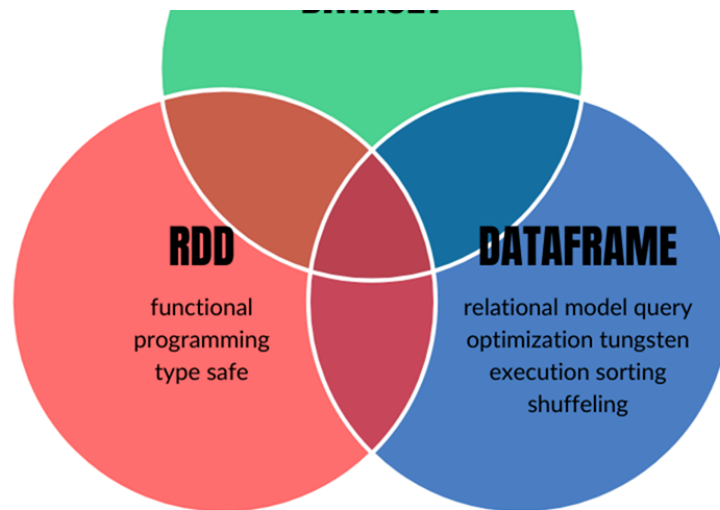
- Allows data to be organized in a **tabular format** with **rows and columns**, similar to a **database table**.

- Utilizes an **optimized execution plan** through the **Catalyst optimizer** for query planning.
- One limitation is the lack of **compile-time type safety**, which means data schema must be known beforehand for data manipulation.
- If you are working with Python, it's recommended to start with **DataFrames** and switch to **RDDs** only if you need more flexibility.

DataSet (A subset of DataFrames):

- Provides the best encoding component and ensures **type safety at compile-time**, unlike **DataFrames**.
- Suitable for scenarios where you need strong **type safety at compile-time** or want to work with typed **JVM objects**.
- You can leverage **Datasets** to take advantage of **Catalyst optimization** and **Tungsten's fast code generation** for improved performance.





Q2) How can you create a DataFrame a) using an existing RDD, and b) from a CSV file?

a) Using an Existing RDD:

To create a **DataFrame** from an existing **RDD**, you can use the **toDF()** function of **PySpark RDD**. By default, the **DataFrame** is constructed with the column names “_1” and “_2” to represent the two columns, since **RDDs** do not have named columns.

Here's how to create a DataFrame from an existing RDD:

```
dfFromRDD1 = rdd.toDF()
dfFromRDD1.printSchema()
```

This will give you a **DataFrame schema** without custom column names:

```
root
|-- _1: string (nullable = true)
|-- _2: string (nullable = true)
```

To assign custom column names to the **DataFrame**, you can pass them as parameters to the **toDF()** function, as shown below:

```
columns = ["language", "users_count"]
dfFromRDD1 = rdd.toDF(columns)
dfFromRDD1.printSchema()
```

The above code snippet will give you a **DataFrame schema** with the specified column names:

```

root
|-- language: string (nullable = true)
|-- users_count: string (nullable = true)

```



b) From a CSV File:

To create a **DataFrame** from a **CSV file**, you can use the **read.csv()** function provided by **PySpark**. Here's how you can do it:

```

dfFromCSV = spark.read.csv("path/to/csvfile.csv", header=True, inferSchema=True)
dfFromCSV.printSchema()

```

This will read the **CSV file** into a **DataFrame**, inferring the schema and using the first row as a header to determine column names.

Q3) Explain the use of StructType and StructField classes in PySpark with examples.

The **StructType** and **StructField** classes in **PySpark** are used to define the schema for a **DataFrame** and create complex columns such as nested struct, array, and map columns.

StructType is a collection of **StructField** objects that determine the column name, column data

type, field nullability, and metadata.

PySpark imports the **StructType** class from **pyspark.sql.types** to describe the data frame's **structure**. The data frames **printSchema()** function displays **StructType** columns as "struct."

To define the columns, **PySpark** offers the **StructField** class from **pyspark.sql.types**, which includes the column name (**String**), column type (**DataType**), nullable column (**Boolean**), and metadata (**MetaData**).

Here is an example showing the use of **StructType** and **StructField** classes in **PySpark**:

```
import pyspark
from pyspark.sql import SparkSession
from pyspark.sql.types import StructType, StructField, StringType, IntegerType
```

```
spark = SparkSession.builder.master("local[1]") \
    .appName('K21Academy') \
    .getOrCreate()

data = [
    ("James", "", "William", "36636", "M", 3000),
    ("Michael", "Smith", "", "40288", "M", 4000),
    ("Robert", "", "Dawson", "42114", "M", 4000),
    ("Maria", "Jones", "", "39192", "F", 4000)
]

schema = StructType([
    StructField("firstname", StringType(), True),
    StructField("middlename", StringType(), True),
    StructField("lastname", StringType(), True),
    StructField("id", StringType(), True),
    StructField("gender", StringType(), True),
    StructField("salary", IntegerType(), True)
])

df = spark.createDataFrame(data=data, schema=schema)
df.printSchema()
df.show(truncate=False)
```

In this example, we define a schema using **StructType** and **StructField** and create a **DataFrame** with sample data. The **printSchema()** the method displays the schema of the **DataFrame** and the **show()** the method displays the data

Q4) What are the different ways to handle row duplication in a PySpark DataFrame?

There are two primary ways to handle row duplication in **PySpark DataFrames**. The **distinct()** function in **PySpark** is used to remove duplicate rows across all columns in a **DataFrame**, while **dropDuplicates()** is used to remove rows based on one or more specific columns.

Here's an example showing how to utilize the **distinct()** and **dropDuplicates()** methods:

First, we need to create a sample **DataFrame**:

```
import pyspark
from pyspark.sql import SparkSession

spark = SparkSession.builder.appName('K21Academy').getOrCreate()

data = [
    ("James", "Sales", 3000),
    ("Michael", "Sales", 4600),
    ("Robert", "Sales", 4100),
    ("Maria", "Finance", 3000),
    ("James", "Sales", 3000),
    ("Scott", "Finance", 3300),
    ("Jen", "Finance", 3900),
    ("Jeff", "Marketing", 3000),
    ("Kumar", "Marketing", 2000),
    ("Saif", "Sales", 4100)
]

columns = ["employee_name", "department", "salary"]
df = spark.createDataFrame(data=data, schema=columns)
df.printSchema()
df.show(truncate=False)
```

```
+-----+-----+-----+
|employee_name|department|salary|
```

James	Sales	3000	
Michael	Sales	4600	
Robert	Sales	4100	
Maria	Finance	3000	
James	Sales	3000	
Scott	Finance	3300	
Jen	Finance	3900	
Jeff	Marketing	3000	
Kumar	Marketing	2000	
Saif	Sales	4100	

The output shows that the record with the employee name “**Robert**” contains duplicate rows. There are two rows with duplicate values in all fields and four rows with duplicate values in the department and salary columns.

Here is the entire code for removing duplicate rows:

```
# Distinct
distinctDF = df.distinct()
print("Distinct count: " + str(distinctDF.count()))
distinctDF.show(truncate=False)
```

```
# Drop duplicates across all columns
df2 = df.dropDuplicates()
print("Distinct count: " + str(df2.count()))
df2.show(truncate=False)

# Drop duplicates based on selected columns
dropDisDF = df.dropDuplicates(["department", "salary"])
print("Distinct count of department and salary: " + str(dropDisDF.count()))
dropDisDF.show(truncate=False)
```

In this code:

- **distinct()** removes all duplicate rows across all columns.
- **dropDuplicates()** removes duplicate rows based on all columns by default.
- **dropDuplicates(["department", "salary"])** removes duplicate rows based on the specified columns "department" and "salary."

Q5) Explain PySpark UDF with the help of an example.

PySpark UDF (User Defined Function) is a crucial aspect of **Spark SQL** and **DataFrame**, allowing users to extend **PySpark's** built-in capabilities. **UDFs** in **PySpark** function similarly to **UDFs** in traditional databases. We write a **Python function**, wrap it in **PySpark SQLudf()**, or register it as a **UDF**, and then use it on **DataFrames** or within **SQL queries** in **PySpark**.

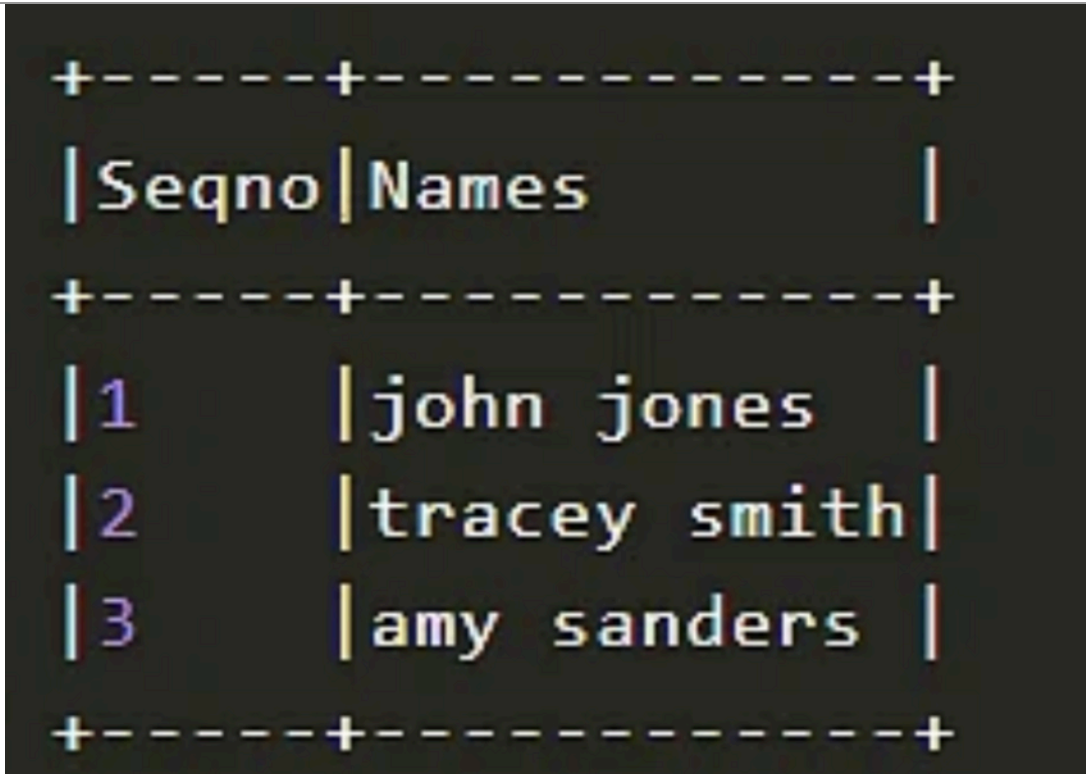
Here's an example of how to create and use a **UDF**:

First, we need to create a sample **DataFrame**:

```
from pyspark.sql import SparkSession

spark = SparkSession.builder.appName('K21Academy').getOrCreate()
```

```
columns = ["Seqno", "Name"]
data = [("1", "john jones"),
        ("2", "tracey smith"),
        ("3", "amy sanders")]
df = spark.createDataFrame(data=data, schema=columns)
df.show(truncate=False)
```



Seqno	Names
1	john jones
2	tracey smith
3	amy sanders

Next, we create a **Python function**. The code below defines the `convertCase()` function, which takes a string parameter and capitalizes the first letter of each word:

```
def convertCase(str):
    resStr = ""
    arr = str.split(" ")
    for x in arr:
```

```
resStr = resStr + x[0:1].upper() + x[1:] + " "  
return resStr.strip()
```

The final step is converting the **Python function** to a **PySpark UDF**. By passing the function to **PySpark SQL udf()**, we can convert the **convertCase()** function to a **UDF**. The **org.apache.spark.sql.functions.udf** package contains this function. Before using this package, we must first import it.

```
from pyspark.sql.functions import udf  
from pyspark.sql.types import StringType  
  
# Converting function to UDF  
convertUDF = udf(lambda z: convertCase(z), StringType())  
  
# Applying the UDF to the DataFrame  
df = df.withColumn("Name", convertUDF(df.Name))  
df.show(truncate=False)
```

This will apply to the **convertCase UDF** to the **“Name”** column of the **DataFrame**, capitalizing the first letter of each word in the names.

PySpark Coding Interview Questions ^



Q6) You have a cluster of ten nodes with each node having 24 CPU cores. The following code works, but it may crash on huge data sets, or at the very least, it may not take advantage of the cluster's full processing capabilities. Which aspect is the most difficult to alter, and how would you go about doing so?

```
def cal(sparkSession: SparkSession): Unit = {  
    val NumNode = 10  
    val userActivityRdd: RDD[UserActivity] = readUserActivityData(sparkSession)  
        .repartition(NumNode)  
    val result = userActivityRdd  
        .map(e => (e.userId, 1L))  
        .reduceByKey(_ + _)  
    result.take(1000)  
}
```

The **repartition** command creates ten partitions regardless of the initial number of partitions. On large datasets, these partitions might become quite large and may outgrow the **RAM** allocated to a single executor. Additionally, each executor can process only one partition at a time, meaning only ten of the 240 available cores (10 nodes with 24 cores each) are utilized.

If the number of partitions is set too high, the scheduler's overhead in handling the partitions increases, reducing performance. This overhead can sometimes exceed the execution time, especially for very small partitions.

The optimal number of partitions is typically between two and three times the number of cores. In this scenario, the ideal number of partitions would be approximately:

$$600 = 10 \times 24 \times 2.5 \quad 600 = 10 \times 24 \times 2.5$$

To modify the code for optimal performance, adjust the number of partitions to better utilize the available cores:

```
def cal(sparkSession: SparkSession): Unit = {
    val NumPartitions = 600
    val userActivityRdd: RDD[UserActivity] = readUserActivityData(sparkSession)
        .repartition(NumPartitions)
    val result = userActivityRdd
        .map(e => (e.userId, 1L))
        .reduceByKey(_ + _)
    result.take(1000)
}
```

By setting the number of partitions to **600**, we ensure that the workload is evenly distributed across the 240 cores, maximizing the cluster's processing capabilities and avoiding memory issues.

Q7) The code below generates two data frames with the following structure: DF1: uid, uName DF2: uid, pageId, timestamp, and eventType. Join the two data frames using code and count the number of events per uName. It should only output for users who have events in the format uName; totalEventCount.

- **DF1:** uid, uName
- **DF2:** uid, pageId, timestamp, eventType

Join the two **DataFrames** using code and count the number of events per **uName**. The output should only include users who have events, in the format **uName; totalEventCount**

Here is the complete code:

```
def calculate(sparkSession: SparkSession): Unit = {
    val UIdColName = "uId"
    val UNameColName = "uName"
    val CountColName = "totalEventCount"
```

```
// Reading user data into a DataFrame
val userDf: DataFrame = readUserData(sparkSession)

// Reading user activity data into a DataFrame
val userActivityDf: DataFrame = readUserActivityData(sparkSession)

// Joining the DataFrames on uId and counting events per uName
val result = userDf
    .join(userActivityDf, UIdColName)
    .groupBy(col(UNameColName))
    .count()
    .withColumnRenamed("count", CountColName)

// Displaying the result
result.show()
}
```

Explanation:

1. **Reading Data:** The `readUserData` and `readUserActivityData` functions are used to read the user data and user activity data into DataFrames (`userDf` and `userActivityDf`, respectively).
2. **Joining DataFrames:** The `join` method is used to join `userDf` and `userActivityDf` on the `uId` column.
3. **Grouping and Counting:** The `groupBy` method is used to group the data by `uName`, and the `count` method counts the number of events for each `uName`.
4. **Renaming Column:** The `withColumnRenamed` method renames the "count" column to `totalEventCount`.
5. **Displaying the Result:** The `show` method displays the final result.

This code ensures that the output includes only users who have events, with the total event count per `uName`.

Q8) Please indicate which parts of the following code will run on the master and which

parts will run on each worker node.

```
val formatter: DateTimeFormatter = DateTimeFormatter.ofPattern("yyyy/MM")

def getEventCountOnWeekdaysPerMonth(data: RDD[(LocalDateTime, Long)]): Array[(String, Long)] = {
    val res = data
        .filter(e => e._1.getDayOfWeek.getValue < DayOfWeek.SATURDAY.getValue)
        .map(mapDateTime2Date)
        .reduceByKey(_ + _)
        .collect()

    res.map(e => (e._1.format(formatter), e._2))
}

private def mapDateTime2Date(v: (LocalDateTime, Long)): (LocalDate, Long) = {
    (v._1.toLocalDate.withDayOfMonth(1), v._2)
}
```

The **driver application (master node)** is responsible for calling this function. The **Directed Acyclic Graph (DAG)** is defined by the assignment to the **res** value, and its execution is initiated by the **collect()** operation. The following details indicate where each part of the code will run:

1. Master Node:

- **Function Invocation:** The function `getEventCountOnWeekdaysPerMonth` is invoked on the master node.
- **DAG Definition:** The assignment to **res** and the definition of the transformations (`filter`, `map`, `reduceByKey`) are part of the DAG creation, which is done on the master node.
- **Result Collection:** The `collect()` method triggers the execution of the DAG and collects the result on the master node.
- **Formatting Results:** The `map` operation on the collected **res** array to format the date and count is performed on the master node.

2. Worker Nodes:

- **Data Transformation and Execution:**
 - The `filter` operation to select events occurring on weekdays.
 - The `map` operation using the `mapDateTime2Date` function.

- The **reduceByKey** operation to aggregate the event counts. These transformations and computations are distributed across the worker nodes.

3. **mapDateTime2Date Function:**

- The logic within the **mapDateTime2Date** method is executed on the worker nodes as part of the **map** transformation.

In summary, the master node handles the function invocation, DAG definition, result collection, and final mapping operation, while the worker nodes handle the distributed data transformations and aggregations.

PySpark Interview Questions for Experienced Data Engineers [^]_{__}

Q9) Under what scenarios are Client and Cluster modes used for deployment?

Cluster Mode:

- **Scenario:** When the client computers are not located near the cluster.
- **Reason:** This mode prevents network delays that would occur in Client mode due to communication between executors and the client machine. Additionally, in Cluster mode, if the client machine goes offline, the operation continues unaffected, as the driver runs within the cluster.

Client Mode:

- **Scenario:** When the client computer is located within the cluster.
- **Reason:** In this mode, there are no network latency issues because the client machine is part of the cluster. Maintenance of the cluster is already managed, so there's no concern about operation loss if the client machine experiences a failure.

In summary, **Cluster mode** is preferred when the client is remote from the cluster to avoid network delays and ensure reliability. **Client mode** is suitable when the client is within the cluster, eliminating network latency concerns and simplifying maintenance.

Q10) How is Apache Spark different from MapReduce?

MapReduce	Apache Spark
Only batch-wise data processing is done using MapReduce.	Apache Spark can handle data in both real-time and batch mode.
The data is stored in HDFS (Hadoop Distributed File System), which takes a long time to retrieve.	Spark saves data in memory (RAM), making data retrieval quicker and faster when needed.
MapReduce is a high-latency framework since it is heavily reliant on disk.	Spark is a low-latency computation platform because it offers in-memory data storage and caching.

Q11) What is meant by Executor Memory in PySpark?

In PySpark, **executor memory** refers to the amount of memory allocated to each executor in a Spark application. Spark executors have a fixed core count and heap size, determined by the application's configuration. The heap size, which represents the memory used by the Spark executor, is controlled by the `spark.executor.memory` property, set using the `--executor-memory` flag.

Each worker node in a Spark cluster is assigned one or more executors, and the executor memory is a measure of the memory utilized by the executors on these worker nodes. This memory is used for executing tasks, storing data, and caching results during the application's runtime.

To summarize, executor memory is the memory allocated to the executors on each worker node in a Spark cluster, crucial for the performance and efficiency of Spark applications.

Q12) How can data transfers be kept to a minimum while using PySpark?

In PySpark, data transfers typically occur during the shuffling process. Minimizing these transfers can lead to faster and more reliable Spark applications. Here are several approaches to reduce data transfers:

1. **Using Broadcast Variables:**
 - Broadcast variables allow the efficient joining of large and small RDDs by distributing a copy of the data to each node, thereby reducing the amount of data shuffled across the network.
2. **Using Accumulators:**
 - Accumulators enable parallel updates to variable values during execution, reducing the need for data transfers and synchronization between nodes.
3. **Avoiding Costly Operations:**
 - Prevent operations that trigger reshuffles, such as **groupByKey**, **distinct**, and **repartition**, whenever possible. Instead, use alternatives like **reduceByKey**, **aggregateByKey**, or **mapPartitions** that are more shuffle-efficient.

By implementing these strategies, you can minimize data transfers and optimize the performance of your PySpark applications.

Interview Questions on PySpark Data Science ^

Q13) What distinguishes Apache Spark from other programming languages?

- **High Data Processing Speed:**
 - Apache Spark achieves very high data processing speeds by minimizing read-write operations to disk. It is approximately 100 times faster for in-memory computations and 10 times faster for disk-based computations.
- **Dynamic Nature:**
 - Spark's dynamic nature is characterized by over 80 high-level operators, which simplify the development of parallel applications.
- **In-Memory Computing Ability:**
 - Spark's in-memory computing capability, enabled by its **Directed Acyclic Graph (DAG)** execution engine, significantly boosts data processing speed. This also facilitates data caching, reducing the time required to retrieve data from disk.
- **Fault Tolerance:**
 - Spark uses **Resilient Distributed Datasets (RDDs)** to support fault tolerance. RDDs are abstractions designed to handle worker node failures without losing data.
- **Stream Processing:**
 - Spark provides real-time stream processing capabilities, addressing the limitation of the previous MapReduce architecture, which could only handle batch-processed data.

Q14) Explain RDDs in detail.

Resilient Distributed Datasets (RDDs) are a fundamental data structure in Apache Spark. They are a collection of fault-tolerant functional units that can be executed in parallel across a cluster. RDDs consist of data fragments that are maintained in memory and distributed across multiple nodes, ensuring that all partitioned data is distributed and consistent.

There are two types of RDDs:

1. Hadoop Datasets:

- These datasets apply a function to each file record in the Hadoop Distributed File System (HDFS) or another file storage system. This allows Spark to leverage existing Hadoop data for parallel processing.

2. Parallelized Collections:

- These are existing collections (such as arrays or lists) that Spark can distribute across multiple nodes to operate in parallel. This enables efficient parallel processing of in-memory data.

RDDs provide the following key features:

- **Fault Tolerance:** RDDs use lineage information to recompute lost data in case of node failures.
- **Immutability:** Once created, RDDs cannot be modified, ensuring consistency and simplifying fault recovery.
- **Lazy Evaluation:** Transformations on RDDs are not executed immediately but are recorded to build a DAG, which is executed only when an action is performed.
- **Partitioning:** Data in RDDs is divided into partitions, which can be processed in parallel across different nodes in the cluster.

These features make RDDs a powerful tool for handling **large-scale data processing** tasks efficiently.

Q15) Mention some of the major advantages and disadvantages of PySpark.

Advantages of PySpark:

1. **Effortless Parallelization:**
 - Writing parallelized code is straightforward, enabling efficient data processing across a cluster without extensive manual intervention.
2. **Error and Synchronization Management:**
 - PySpark automatically handles synchronization points and tracks errors, simplifying debugging and ensuring reliable execution.
3. **Rich Library of Built-in Algorithms:**
 - PySpark comes with a wide range of useful built-in algorithms for machine learning, data manipulation, and analysis, making it a powerful tool for various data processing tasks.

Disadvantages of PySpark:

1. **Complexity in Managing MapReduce Issues:**
 - Handling issues with MapReduce can be challenging and may require in-depth knowledge of the underlying framework.
2. **Inefficiency Compared to Alternative Paradigms:**
 - PySpark can be less efficient compared to other programming paradigms, particularly for certain types of data processing tasks that do not benefit from distributed computing.

Advanced PySpark Interview Questions and Answers ^

Q16) Discuss PySpark SQL in detail.

PySpark SQL is a structured data library for Spark that provides more detailed information about data structure and operations than the PySpark RDD API. It introduces a powerful programming paradigm known as **DataFrame**.

DataFrame: A DataFrame is an immutable distributed columnar data collection that can process large volumes of structured data (like relational databases) and semi-structured data (such as JSON). Once a DataFrame is created, you can interact with the data using SQL syntax and queries.

Using PySpark SQL: To use PySpark SQL, the first step is to create a temporary table on the DataFrame using the `createOrReplaceTempView()` function. This temporary table is accessible throughout the SparkSession using the `sql()` method. The temporary table can be deleted by ending the SparkSession.

Example of PySpark SQL:

```
import findspark
findspark.init()
import pyspark
from pyspark.sql import SparkSession
```

```
# Create a SparkSession
spark = SparkSession.builder.getOrCreate()

# Create a DataFrame using SQL syntax
df = spark.sql("SELECT 'spark' AS hello")

# Show the DataFrame
df.show()
```

In this example:

- We initialize PySpark using **findspark**.
- We create a **SparkSession**, which is the entry point for using PySpark SQL.
- We create a DataFrame using an SQL query.
- We display the DataFrame using the **show()** method.

PySpark SQL makes it easy to work with structured and semi-structured data using familiar SQL syntax, enhancing the efficiency and flexibility of data processing in Spark.

Q17) Explain the different persistence levels in PySpark.

Persisting (or caching) a dataset in memory is one of PySpark's most essential features. The different levels of persistence in PySpark are as follows:

Level	Purpose
MEMORY_ONLY	Stores deserialized Java objects in the JVM. It is the default persistence level in PySpark.
MEMORY_AND_DISK	Stores RDD as deserialized Java objects. If the RDD is too large to fit in memory, it saves the partitions that don't fit to disk and reads them as needed.
MEMORY_ONLY_SER	Stores RDD as serialized Java objects. This level saves more space when using fast serializers but requires more CPU capacity to read the RDD.
MEMORY_AND_DISK_SER	Similar to MEMORY_ONLY_SER, it stores partitions on disk instead of recomputing them each time they're needed.
DISK_ONLY	Stores RDD partitions only on the disk.

Level	Purpose
MEMORY_ONLY_2,	Function the same as MEMORY_ONLY and MEMORY AND DISK but replicate each partition on two cluster

MEMORY_AND_DISK_2	Stores data in memory on all nodes.
OFF_HEAP	Requires off-heap memory to store RDD.

Q18) What do you mean by checkpointing in PySpark?

A streaming application must be available 24/7 and be resilient to errors external to the application code, such as system failures or JVM crashes. **Checkpointing** is a process that enhances the fault tolerance of streaming applications by saving data and metadata to a checkpoint directory.

There are two types of checkpointing in PySpark:

1. **Metadata Checkpointing:**

- This type of checkpointing saves the information that defines the streaming computation to a fault-tolerant storage system like HDFS. It helps recover data in the event of a failure of the streaming application's driver node.

2. **Data Checkpointing:**

- This involves saving the generated RDDs to a reliable storage location. Stateful computations that combine data from different batches often require this type of checkpointing to ensure that data is not lost during failures.

Checkpointing makes streaming applications more robust by ensuring that both the computation logic (metadata) and the data itself are preserved, allowing the application to recover from various types of failures efficiently.

Q19) Define the role of Catalyst Optimizer in PySpark.

The **Catalyst optimizer** is a crucial component of Apache Spark, significantly enhancing the performance of structural queries expressed in SQL or via the DataFrame/Dataset APIs. It reduces program runtime and lowers costs by optimizing the execution plans for queries.

The Spark **Catalyst optimizer** supports two main types of optimization:

1. **Rule-Based Optimization:**

- This involves a set of predefined rules that dictate how to transform and execute the query. These rules ensure the query is executed in the most efficient manner possible.

2. **Cost-Based Optimization:**

- This involves generating multiple execution plans based on rules and then calculating the cost of each plan. The optimizer selects the plan with the lowest cost, ensuring optimal resource usage and performance.

In addition to query optimization, the **Catalyst optimizer** addresses various Big Data challenges, such as handling semi-structured data and supporting advanced analytics. By doing so, it plays a pivotal role in making Spark a powerful tool for large-scale data processing.

PySpark Scenario-Based Interview Questions for Experienced Professionals ^

Q20) The given file has a delimiter ~|. How will you load it as a spark DataFrame?

Important: Instead of using `sparkContext(sc)`, use `sparkSession (spark)`.

Input File

```
Name ~|Age
Azarudeen, Shahul~|25
Michel, Clarke ~|26
Virat, Kohli ~|28
Andrew, Simond ~|37
George, Bush~|59
Flintoff, David ~|12
```

```
import findspark
findspark.init()
from pyspark.sql import SparkSession
```

```
# Initialize SparkSession
spark = SparkSession.builder.master("local").appName("scenario based").getOrCreate()

# Read the file as a text file
df = spark.read.text("input.csv")

# Display the DataFrame
df.show(truncate=False)

# Extract the header
header = df.first()[0]
schema = header.split("~|")

# Filter out the header and split the remaining lines by the delimiter
df_input = df.filter(df['value'] != header).rdd.map(lambda x: x[0].split("~|")).toDF(sc

# Display the DataFrame with the correct schema
df_input.show(truncate=False)
```

Explanation:

1. **Initialize SparkSession:** Use **SparkSession** instead of **sparkContext**.
2. **Read the File:** Read the file as a text file into a DataFrame.
3. **Extract Header:** Extract the header from the first row and split it using the delimiter **~|** to create the schema.
4. **Filter and Split:** Filter out the header row from the DataFrame and split the remaining rows by the delimiter **~|**.
5. **Create DataFrame:** Create a new DataFrame with the correct schema.
6. **Display the DataFrame:** Show the DataFrame with the correct schema.

This ensures the file is correctly loaded into a Spark DataFrame, with the specified delimiter.

Q21) How will you merge two files – File1 and File2 – into a single DataFrame if they have different schemas?

File 1:

```
Name|Age
Azarudeen, Shahul|25
Michel, Clarke|26
Virat, Kohli|28
Andrew, Simond|37
```

File 2:

```
Name|Age|Gender
Rabindra, Tagore|32|Male
Madona, Laure|59|Female
Flintoff, David|12|Male
Ammie, James|20|Female
```

```
import findspark
findspark.init()
from pyspark.sql import SparkSession
from pyspark.sql.functions import lit
```



```
from pyspark.sql.types import StructType, StructField, StringType

# Initialize SparkSession
spark = SparkSession.builder.master("local").appName('Modes of DataFrameReader').getOrCreate()

# Read File 1
df1 = spark.read.option("delimiter", "|").csv('input.csv', header=True)

# Read File 2
df2 = spark.read.option("delimiter", "|").csv("input2.csv", header=True)

# Add missing column to df1
df1 = df1.withColumn("Gender", lit(None).cast(StringType()))

# Union the DataFrames
df_combined = df1.unionByName(df2, allowMissingColumns=True)
df_combined.show()

# For Union with defined schema
schema = StructType([
    StructField("Name", StringType()
```

Explanation:

1. **Initialize SparkSession:** Start by initializing a **SparkSession**.
2. **Read the Files:** Read both files using the **spark.read.csv** method, specifying the delimiter as **|** and including headers.
3. **Add Missing Column:** For **df1**, add the missing "Gender" column with **None** values to match the schema of **df2**.
4. **Union the DataFrames:** Use **unionByName** with **allowMissingColumns=True** to merge the DataFrames with different schemas.
5. **Union with Defined Schema:** Define a schema that includes all expected columns, read the files using this schema, and perform the union again to ensure consistent structure.

This ensures that both files are merged into a single DataFrame correctly, even if they have different

schemas.

Q22) Examine the following file, which contains some corrupt/bad data. What will you do with such data, and how will you import them into a Spark DataFrame?

```
Emp_no, Emp_name, Department
101, Murugan, HealthCare
Invalid Entry, Description: Bad Record entry
102, Kannan, Finance
103, Mani, IT
Connection lost, Description: Poor Connection
104, Pavan, HR
Bad Record, Description: Corrupt record
```

```
import findspark
findspark.init()
from pyspark.sql import SparkSession
from pyspark.sql.types import StructType, StructField, StringType
```

```
# Initialize SparkSession
spark = SparkSession.builder.master("local").appName("Modes of DataFrameReader").getOrCreate()

# Define the schema
schema = StructType([
    StructField("Emp_no", StringType(), True),
    StructField("Emp_name", StringType(), True),
    StructField("Department", StringType(), True),
])

# Read the file and drop malformed records
df = spark.read.option("mode", "DROPMALFORMED").csv('input1.csv', header=True, schema=schema)

# Show the DataFrame
df.show()
```

Explanation:

1. **Initialize SparkSession:** Start by initializing a **SparkSession**.
2. **Define the Schema:** Define a schema that matches the expected structure of the data, including the column names and data types.
3. **Read the File:** Read the file using the **spark.read.csv** method with the **option("mode", "DROPMALFORMED")** to drop any malformed records.
4. **Show the DataFrame:** Display the DataFrame to verify that the data has been imported correctly, with corrupt/bad data excluded.

By using the **DROPMALFORMED** option, we ensure that only well-formed records are included in the DataFrame, effectively handling the corrupt/bad data.

Capgemini PySpark Interview Questions ^

Q23) What is SparkConf in PySpark? List a few attributes of SparkConf.

SparkConf is a configuration class in PySpark that is used to set up and configure the settings needed to run a Spark application, either locally or on a cluster. In other words, it provides various settings and parameters required for running a Spark application.

Here are some of the most important features and attributes of **SparkConf**:

1. **set(key, value):**
 - This method is used to set a configuration property.
 - Example: `conf.set("spark.executor.memory", "2g")`
2. **setSparkHome(value):**
 - This method allows you to specify the directory where Spark is installed on worker nodes.
 - Example: `conf.setSparkHome("/path/to/spark/home")`
3. **setAppName(value):**
 - This method is used to specify the name of the application.
 - Example: `conf.setAppName("MySparkApp")`
4. **setMaster(value):**
 - This method sets the master URL, which determines the cluster manager to connect to.
 - Example: `conf.setMaster("local[*]")` for local mode or

`conf.setMaster("spark://master:7077")` for cluster mode.

5. **get(key, defaultValue=None):**

- This method retrieves the configuration value for a given key. If the key does not exist, it returns the default value if provided.
- Example: `conf.get("spark.executor.memory", "1g")`

By using these attributes, **SparkConf** allows you to customize and optimize the execution of your Spark application according to your specific requirements.

Q24) What are the various types of Cluster Managers in PySpark?

PySpark supports the following **cluster managers**:

1. **Standalone:**

- A simple cluster manager that comes built-in with Spark. It simplifies the process of setting up a Spark cluster.

2. **Apache Mesos:**

- A cluster manager that can run a variety of distributed applications, including Hadoop MapReduce and PySpark applications. It provides resource isolation and sharing across distributed applications.

3. **Hadoop YARN:**

- The resource management layer of Hadoop 2, allows Spark to run on Hadoop clusters. It manages resources and scheduling of Spark applications alongside other Hadoop applications.

4. **Kubernetes:**

- An open-source platform designed for automating the deployment, scaling, and management of containerized applications. Spark can run on Kubernetes to leverage its container orchestration capabilities.

5. **local:**

- Not a cluster manager per se, but it is worth mentioning. The "local" master mode is used to run Spark on a single machine, such as a laptop or desktop, for development and

testing purposes.

Each of these **cluster managers** provides different benefits and use cases, allowing Spark to be flexible and adaptable to various environments and workloads.

Q25) Explain how Apache Spark Streaming works with receivers.

Receivers are special objects in Apache Spark Streaming designed to consume data from various data sources and move it into Spark for processing. These receivers run as long-running tasks on different executors within a Spark Streaming context, allowing continuous data ingestion.

There are two types of receivers in Spark Streaming:

1. **Reliable Receiver:**

- This type of receiver acknowledges the data sources after the data has been successfully received and stored in Apache Spark Storage. It ensures data reliability and fault tolerance by confirming that the data has been processed correctly.

2. **Unreliable Receiver:**

- This type of receiver does not acknowledge the data sources when receiving or replicating data in Apache Spark Storage. As a result, it may lead to data loss in case of failures

since it does not confirm the successful reception and storage of data.

Receivers play a crucial role in Spark Streaming by enabling the continuous flow of data from sources such as Kafka, Flume, or TCP sockets into Spark for real-time processing and analysis.

Download the Full Pyspark Interview Guide

Mastering these 60+ essential Pyspark interview questions will enhance your confidence and preparation for your upcoming Pyspark job interviews.

[maxbutton id="5" url="https://k21academy.com/data07" text="Download Now"]

Conclusion ^

With these top **PySpark interview questions and answers** at your disposal, you'll be more confident and prepared for your next job interview. Whether the discussion revolves around **PySpark DataFrames**, **coding challenges**, **deployment scenarios**, or **advanced concepts**, this comprehensive guide ensures you're well-equipped. Enter your interview ready to demonstrate your **expertise in the dynamic field of PySpark**.

Frequently Asked Questions on PySpark

Q1) Is PySpark the same as Spark?

No, PySpark is not the same as Spark. PySpark is a Python API for Apache Spark, enabling the development of Spark applications using Python.

Q2) What is PySpark, and how does it work?

PySpark is a Python API for Apache Spark that allows you to develop Spark applications using Python. It includes the PySpark shell for interactive data analysis in a distributed environment. PySpark

supports most of Spark's capabilities, including Spark SQL, DataFrame, Streaming, MLlib (Machine Learning), and Spark Core.

Q3) Is PySpark a Big Data tool? Does PySpark require Spark?

Yes, PySpark is a powerful Big Data tool that requires Apache Spark to function. It is a Python library for Spark, designed to run Python applications using Spark's features.

Q4) Is PySpark easy to learn?

PySpark is relatively easy to learn for individuals who have a basic understanding of Python, Java, or other programming languages.

Q5) How long does it take to learn PySpark?

If you have a solid foundation in object-oriented and functional programming, you can learn the basics of the Spark Core API within a week.

Q6) What is the best way to learn PySpark? Is PySpark a framework?

PySpark is an open-source framework that provides a Python API for Spark. To learn PySpark, start with Python, SQL, and Apache Spark. Additionally, gain practical experience by working on real-world projects.

Related References

- [Microsoft Azure Data on Cloud Job-Oriented Step-by-Step Activity Guides.](https://k21academy.com/microsoft-azure/data-engineer/azure-data-science-and-data-)
(<https://k21academy.com/microsoft-azure/data-engineer/azure-data-science-and-data->

[engineering-certifications-dp-900-vs-dp-100-vs-dp-200-dp-201/](#)

- [Azure Data Factory For Beginners \(https://k21academy.com/microsoft-azure/data-engineer/azure-data-factory/\)](#)
- [Top 100+ Data Modelling Interview Questions \(https://k21academy.com/data-engineering/100-data-modelling-interview-questions/\)](#)
- [Top 50+ ETL Interview Questions \(https://k21academy.com/data-engineering/top-50-etl-interview-questions-and-answers-for-data-professionals/\)](#)
- [Azure Synapse Analytics \(Azure SQL Data Warehouse\) \(https://k21academy.com/microsoft-azure/data-engineer/azure-synapse-analytics/\)](#)
- [Azure SQL Database | All You Need to Know About Azure SQL Services \(https://k21academy.com/microsoft-azure/data-engineer/azure-sql-database-azure-data-engineer-associate-dp-200-dp-201/\)](#)

Next Task For You

Our Azure Data Engineer training program will cover **50 Hands-On Labs**. If you want to begin your journey towards becoming a **Microsoft Certified: Azure Data Engineer Associate** check out our **FREE CLASS** (https://k21academy.com/free-class-azure-data-engineer-certification/?utm_source=content_upgrade&utm_medium=referral&utm_campaign=dp20302_july24).

(https://k21academy.com/azure-data-engineer-certification-free-class/?utm_source=content_upgrade&utm_medium=referral&utm_campaign=dp20302_july24).

Site Map

[Courses](#)

(<https://k21academy.com/>

[all-courses/](#)).

[Blog](#)

(<https://k21academy.com/>

[blog/](#)).

[\(https://www.facebook.com/k21academy/\)](https://www.facebook.com/k21academy/)

(<http://k21academy.com/affiliate/>).

[Careers](#)

(<https://www.linkedin.com/company/k21academy/>).

(<http://k21academy.com/careers/>).

[\(https://www.instagram.com/k21academy/\)](https://www.instagram.com/k21academy/)

[Contact Us](#)

(<https://k21academy.com/contact-us/>).

[Terms and Conditions](#)

(<https://k21academy.com/terms-and-conditions/>).

[Privacy Policy](#)

(<https://k21academy.com/privacy-policy/>).

[Privacy Policy](#)

(<https://k21academy.com/privacy-policy/>).

[Privacy Policy](#)

Courses

[Docker and Kubernetes](#)

[Job Oriented Program](#)

(<https://k21academy.com/docker-kubernetes-training/>).

[docker-kubernetes-](#)

[training/](#)

[AWS Job Oriented](#)

[Program](#)

(<https://k21academy.com/awsjob-oriented-program/>).

[awsjob-oriented-program/](#)).

[Azure Job Oriented](#)

[Program](#)

(<https://k21academy.com/azurejob-oriented-program/>).

[azurejob-oriented-](#)

[program/](#)).

[Azure Data Job Oriented](#)

[Program](#)

(<https://k21academy.com/azure-data-cloud-job-program/>).

[azure-data-cloud-job-](#)

[program/](#)).

[DevOps Job Oriented](#)

[Program](#)

(<https://k21academy.com/devops-job-program/>).

[devops-job-program/](#)).

[Oracle Cloud Job](#)

[Oriented Program](#)

(<https://k21academy.com/oracle-cloud-job-program/>).

[oracle-cloud-job-program/](#)).

[Terraform Job Oriented](#)

(<https://k21academy.com/terraform-job-oriented-program/>).

Get in touch with us

8 Magnolia Pl,
Harrow HA2 6DS,
United Kingdom

Phone:

US: +1 530 264

8480 (tel:

+15302648480).

Email:

contact@k21academy.com

(mailto:contact@k21academy.com)

[1academy.com](mailto:contact@k21academy.com)).

(/).

"Learn Cloud, Data & AI From Experts"



(<https://www.facebook.com/k21academy/>).



(<https://twitter.com/k21academy/>).



(<https://www.linkedin.com/company/k21academy/>).



(<https://www.instagram.com/k21academy/>).



(<https://www.youtube.com/k21academy/>).

[nfra-automation-](#)
[certification-terraform-](#)
[training/](#)

Copyrights © 2012-2025, K21Academy. All Rights Reserved