1. Create RDD from a List

python Copy code

```python
from pyspark import SparkContext

sc = SparkContext()
data = [1, 2, 3, 4, 5]
rdd_from_list = sc.parallelize(data)
```

- **Explanation:** `parallelize` converts a Python list into an RDD.

2. Create RDD from a Text File

python Copy code

```python
rdd_from_text = sc.textFile("path/to/textfile.txt")
```

- **Explanation:** `textFile` reads a file and creates an RDD of lines.

3. Sorting and Extracting from RDD

python Copy code

```python
sorted_rdd = rdd_from_list.sortBy(lambda x: x)  # Sort RDD in ascending order
first_3 = sorted_rdd.take(3)  # Get first 3 elements
```

- **Explanation:** `sortBy` sorts the RDD, and `take` retrieves the top N elements.

4. Save RDD as a Text File

python Copy code

```python
sorted_rdd.saveAsTextFile("path/to/save/textfile")
```

- **Explanation:** `saveAsTextFile` writes the RDD contents to a text file.

5. Create DataFrame from RDD

python Copy code

```python
from pyspark.sql import SparkSession

spark = SparkSession.builder.appName("example").getOrCreate()
rdd = sc.parallelize([(1, "Alice"), (2, "Bob")])
df_from_rdd = spark.createDataFrame(rdd, ["ID", "Name"])
```

- **Explanation**: `createDataFrame` converts an RDD to a DataFrame.

6. Read CSV in DataFrame

python Copy code

```python
df_csv = spark.read.csv("path/to/file.csv", header=True, inferSchema=True)
```

- **Explanation**: Reads a CSV file into a DataFrame with schema inference and headers.

7. Write DataFrame

python Copy code

```python
df_csv.write.csv("path/to/output.csv", header=True)
```

- **Explanation**: Writes the DataFrame as a CSV file.

8. Select Columns from DataFrame

python Copy code

```python
df_selected = df_csv.select("column1", "column2")
```

- **Explanation**: `select` retrieves specific columns from the DataFrame.

9. Add Columns to DataFrame

python Copy code

```python
from pyspark.sql.functions import lit

df_with_new_col = df_csv.withColumn("new_column", lit("default_value"))
```

- **Explanation:** `withColumn` adds a new column with a default value.

10. Drop Columns from DataFrame

python Copy code

```python
df_dropped = df_csv.drop("column1")
```

- **Explanation:** `drop` removes columns from a DataFrame.

11. Sort Data in DataFrame

python Copy code

```python
df_sorted = df_csv.orderBy("column1", ascending=False)
```

- **Explanation:** `orderBy` sorts the DataFrame by a specific column.

12. Filter Data in DataFrame

python Copy code

```python
df_filtered = df_csv.filter(df_csv["column1"] > 100)
```

- **Explanation:** `filter` applies conditions to rows.

13. Remove Duplicates in DataFrame

python

```python
df_deduped = df_csv.dropDuplicates(["column1"])
```

- **Explanation:** `dropDuplicates` removes rows with duplicate values in the specified columns.

14. Combine DataFrames

python

```python
df_combined = df_csv.union(df_csv_2)
```

- **Explanation:** `union` combines two DataFrames with the same schema.

15. Fill Null Values in DataFrame

python

```python
df_filled = df_csv.fillna({"column1": 0})
```

- **Explanation:** `fillna` replaces `null` values in specific columns.

16. Pattern-Based Filters in DataFrame

python

```python
df_filtered = df_csv.filter(df_csv["column2"].rlike("pattern"))
```

- **Explanation:** `rlike` filters rows based on regex patterns.

17. Add Columns Based on Conditions

python Copy code

```python
k.sql.functions import when

nal = df_csv.withColumn("new_col", when(df_csv["column1"] > 100, "high").otherwise("low"))
```

- **Explanation:** `when` adds a new column with conditional logic.

18. Case Conversion in DataFrame

python Copy code

```python
from pyspark.sql.functions import lower, upper

df_lower = df_csv.withColumn("lower_col", lower(df_csv["column2"]))
df_upper = df_csv.withColumn("upper_col", upper(df_csv["column2"]))
```

- **Explanation:** `lower` and `upper` convert text to lowercase or uppercase.

19. Get Top and Bottom Records from DataFrame

python Copy code

```python
top_5 = df_csv.orderBy("column1", ascending=False).limit(5)
bottom_5 = df_csv.orderBy("column1").limit(5)
```

- **Explanation:** `orderBy` sorts the DataFrame, and `limit` retrieves the top/bottom N rows.

20. Aggregation on DataFrame

python Copy code

```python
from pyspark.sql.functions import avg, sum

df_agg = df_csv.agg(avg("column1"), sum("column2"))
```

- **Explanation:** `agg` performs aggregation on DataFrame columns.

21. Aggregation with GroupBy on DataFrame

```python
df_grouped = df_csv.groupBy("column1").agg(sum("column2"))
```

- **Explanation:** `groupBy` groups rows and `agg` applies aggregation functions.

22. Pivot on DataFrame

```python
df_pivot = df_csv.groupBy("column1").pivot("column2").sum("column3")
```

- **Explanation:** `pivot` changes unique values of one column into multiple columns.

23. Unpivot on DataFrame

PySpark doesn't have a direct unpivot function. You can use `selectExpr` with `stack` for unpivoting:

```python
df_unpivot = df_csv.selectExpr("column1", "stack(2, 'col2', col2_value, 'col3', col3_value
```

- **Explanation:** `stack` is used for unpivoting, where '2' is the number of columns being unpivoted.

23. Unpivot on DataFrame

PySpark doesn't have a direct unpivot function. You can use `selectExpr` with `stack` for unpivoting:

```python
tExpr("column1", "stack(2, 'col2', col2_value, 'col3', col3_value) as (attribute, value)")
```

- **Explanation:** `stack` is used for unpivoting, where '2' is the number of columns being unpivoted.

INTERVIEW QUESTIONS AND THEIR ANSWERS:

1. What is PySpark?

Answer:
PySpark is the Python API for Apache Spark, an open-source, distributed computing system that enables fast and general-purpose cluster computing. PySpark allows you to harness the power of Spark with Python, enabling large-scale data processing, machine learning, and real-time data analytics.

2. What is an RDD? How do you create an RDD in PySpark?

Answer:

- **RDD** stands for Resilient Distributed Dataset, the fundamental data structure in Spark. It is an immutable distributed collection of objects that can be processed in parallel across a cluster.
- **Creating an RDD**:
  - From a Python collection:

    python Copy code

    ```python
    rdd = sc.parallelize([1, 2, 3, 4, 5])
    ```

  - From a text file:

    python Copy code

    ```python
    rdd = sc.textFile("path/to/file.txt")
    ```

3. What is a DataFrame in PySpark?

Answer:
A **DataFrame** in PySpark is a distributed collection of data organized into named columns, similar to a table in a relational database or a data frame in pandas. It is built on top of RDDs and is optimized for performance using Spark's Catalyst optimizer.

---

4. What are the key differences between RDDs and DataFrames in PySpark?

Answer:

- **Performance**: DataFrames are faster due to Catalyst optimization, while RDDs are slower as they lack optimizations.
- **API**: DataFrames have a higher-level, SQL-like API, whereas RDDs use a low-level, functional API.
- **Schema**: DataFrames store schema information, whereas RDDs do not.
- **Ease of Use**: DataFrames are easier to use, especially for SQL-style queries, while RDDs require more code for the same operations.

5. What is the difference between `map()` and `flatMap()` in PySpark?

Answer:

- `map()`: Applies a function to each element of the RDD and returns a new RDD with each transformed element.
  - Example:

```python
rdd = sc.parallelize([1, 2, 3])
rdd.map(lambda x: [x, x + 1]).collect()  # Output: [[1, 2], [2, 3], [3, 4]]
```

- `flatMap()`: Similar to `map()` but flattens the result by removing nested lists, returning a flattened RDD.
  - Example:

```python
rdd = sc.parallelize([1, 2, 3])
rdd.flatMap(lambda x: [x, x + 1]).collect()  # Output: [1, 2, 2, 3, 3, 4]
```

6. What is lazy evaluation in PySpark?

Answer:
Lazy evaluation means that transformations on RDDs and DataFrames are not executed immediately. Instead, Spark builds up a logical execution plan. The actual computation is performed only when an action (like `collect()`, `count()`, or `saveAsTextFile()`) is triggered. This helps optimize the overall execution by reducing unnecessary computations and combining transformations.

7. Explain the difference between transformations and actions in PySpark.

Answer:

- **Transformations**: These are operations that return a new RDD/DataFrame, such as `map()`, `filter()`, and `groupBy()`. Transformations are lazily evaluated.
- **Actions**: These trigger the execution of transformations and return a result, such as `collect()`, `count()`, `first()`, `saveAsTextFile()`. Actions force Spark to actually process the data.

8. What is the use of the `persist()` and `cache()` methods in PySpark?

Answer:
Both `persist()` and `cache()` are used to store an RDD or DataFrame in memory to improve performance when it is accessed multiple times.

- `cache()`: By default, stores data in memory (default storage level: `MEMORY_ONLY`).
- `persist()`: Allows you to specify the storage level (e.g., memory, disk, or both).
  Example:

  python Copy code

  ```python
  rdd.persist(StorageLevel.DISK_ONLY)
  rdd.cache()  # Equivalent to rdd.persist(StorageLevel.MEMORY_ONLY)
  ```

9. How do you perform joins in PySpark DataFrames?

Answer: Joins are used to combine two DataFrames based on a common key or condition. PySpark supports various join types such as inner, outer, left, and right joins.

Example of an inner join:

python Copy code

```python
df1.join(df2, df1["key"] == df2["key"], "inner").show()
```

- **Explanation**: Joins `df1` and `df2` on the "key" column with an inner join.

10. What is the `repartition()` function and why is it used?

Answer:
`repartition()` is used to increase or decrease the number of partitions in an RDD or DataFrame. It is often used when you need to re-distribute data across partitions to balance the workload or improve parallelism.

Example:

python Copy code

```python
df = df.repartition(10)
```

- **Explanation**: Repartitions the DataFrame into 10 partitions for better parallel processing.

11. How do you handle missing data in PySpark DataFrames?

Answer: You can handle missing data in several ways:

- **Drop rows with nulls**:

  python Copy code

  ```python
  df.dropna().show()
  ```

- **Fill null values with a default value**:

  python Copy code

  ```python
  df.fillna(0).show()
  ```

12. What are UDFs in PySpark and how are they used?

Answer:
A **UDF (User-Defined Function)** is a custom function that can be applied to columns in PySpark DataFrames. UDFs allow the use of custom Python code but should be avoided unless necessary, as they can slow down performance (since they bypass Spark's Catalyst Optimizer).

Example:

python Copy code

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

def my_func(value):
    return value.upper()

my_udf = udf(my_func, StringType())
df = df.withColumn("new_col", my_udf(df["col"]))
```

13. What are the different persistence/storage levels in PySpark?

Answer:
PySpark offers multiple storage levels for persisting RDDs and DataFrames:

- **MEMORY_ONLY**: Stores data in memory. If data doesn't fit, recomputation is required.
- **MEMORY_AND_DISK**: Stores data in memory; if it overflows, it writes to disk.
- **DISK_ONLY**: Stores data only on disk.
- **MEMORY_ONLY_SER**: Stores serialized objects in memory, reducing space.
- **OFF_HEAP**: Stores data in off-heap memory (requires additional configuration).

14. What are the optimizations done in the Spark Catalyst Optimizer?

Answer: The **Catalyst Optimizer** is the key component in Spark SQL that optimizes queries by:

- Rewriting query plans for better performance.
- Eliminating redundant operations.
- Predicate pushdown to filter data as early as possible.
- Optimizing joins, aggregations, and projections.
- Using Tungsten for code generation and memory management.

15. **What is the difference between narrow and wide transformations in PySpark?**

Answer:

- **Narrow transformations**: Data is only shuffled within a single partition, such as in `map()`, `filter()`. These are more efficient.
- **Wide transformations**: Data is shuffled across multiple partitions, such as in `groupBy()`, `join()`. These transformations are more expensive as they require a shuffle.