Both **SparkContext** and **SparkSession** are essential components in Spark, but they serve different purposes.

1. SparkContext (Older API)

Definition

- **SparkContext** was the entry point for interacting with Spark in **RDD-based APIs (before Spark 2.0)**.
- It was used to create **RDDs (Resilient Distributed Datasets)** and interact with the Spark cluster.

Usage

- When you initialize Spark, `SparkContext` is created automatically inside the `SparkSession`.
- If you are using **older versions of Spark (before 2.0)**, you have to create a `SparkContext` explicitly.

Example

```
python
CopyEdit
from pyspark import SparkContext

sc = SparkContext("local", "MyApp")  # Creating SparkContext
rdd = sc.parallelize([1, 2, 3, 4, 5])  # Creating an RDD
print(rdd.collect())  # Output: [1, 2, 3, 4, 5]
```

Limitations

- Only supports RDD-based APIs.
- Doesn't support **DataFrames** and **Datasets** directly.
- Requires a separate SQLContext, HiveContext, etc., for SQL-based operations.

2. SparkSession (Unified API, Introduced in Spark 2.0)

Definition

- `SparkSession` is the **new unified entry point** to Spark functionalities.
- It **internally contains SparkContext**, along with `SQLContext`, `HiveContext`, and `StreamingContext`.
- It allows operations on **RDDs, DataFrames, and Datasets**.

Usage

- Introduced in **Spark 2.0**, `SparkSession` simplifies working with Spark.
- Supports **both DataFrames and RDDs**.

```
from pyspark.sql import SparkSession

spark = SparkSession.builder.appName("MyApp").getOrCreate()  # Creating SparkSession
df = spark.createDataFrame([(1, "Alice"), (2, "Bob")], ["id", "name"])  # Creating DataFrame
df.show()
```

Advantages of SparkSession

✅ **Unified API** – Works with RDDs, DataFrames, and Datasets.

✅ **Easier Configuration** – No need to create separate `SQLContext`, `HiveContext`, etc.

✅ **Optimized Performance** – Uses **Catalyst Optimizer** for SQL and DataFrame operations.

Key Differences: SparkContext vs SparkSession

Feature	SparkContext	SparkSession
Introduced In	Spark 1.x	Spark 2.x
Purpose	Entry point for RDDs	Entry point for DataFrames, Datasets, RDDs
API Type	Low-level API	High-level API
SQL Support	Requires separate `SQLContext`	Built-in SQL support
DataFrame Support	Not directly supported	Fully supported
Streaming Support	Requires `StreamingContext`	Built-in support

Which One to Use?

- **Use SparkSession** 🏆 (Recommended) → If you are using Spark **2.0 or later**.
- **Use SparkContext** → Only if working with **low-level RDDs** or **older Spark versions**.

Example: Getting SparkContext from SparkSession

If needed, you can still access `SparkContext` from `SparkSession`:

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

sc = spark.sparkContext  # Accessing SparkContext from SparkSession

print(sc.appName)


# Output: MyApp
```

RDD next page

```
num = sc.parallelize([5,5,4,3,2,9,2,4,5,6,7,8,9], 9)
```

Understanding the Command in Real-Time Scenarios

Breakdown of the Command:

1. **`sc.parallelize(data, numPartitions)`**
    a. `sc`: Refers to the **SparkContext**, which is the entry point for Spark functionality.
    b. `parallelize([5,5,4,3,2,9,2,4,5,6,7,8,9], 9)`:
        i. **Creates an RDD (Resilient Distributed Dataset)** from the given list `[5,5,4,3,2,9,2,4,5,6,7,8,9]`.
        ii. **Divides the data into 9 partitions** for parallel processing.

Real-Time Meaning and Use Case

The command **distributes** the list of numbers **across 9 partitions**, allowing Spark to process them in parallel.

Real-Time Scenario Example:

Imagine you're **analyzing sensor readings from 9 different IoT devices** in a factory. Each device generates a **small batch of temperature readings** (like the numbers in the list). By partitioning the data across **9 partitions**, Spark can:

- Process the readings **in parallel**.
- Improve **computation speed**.
- Enable **fault tolerance** (if a partition fails, Spark can recompute only the affected partition).

Example Use Case:

Assume you want to **find the average temperature** from sensor readings.

```
avg_temp = num.mean()
print(avg_temp)
```

This will compute the **mean of all sensor readings efficiently** by leveraging Spark's distributed computing.

Key Takeaways:

1. **Parallel Processing:** The data is split across **9 partitions** for parallel execution.
2. **Efficient Computation:** Allows Spark to handle large datasets efficiently.
3. **Scalability:** If new sensors are added, Spark can dynamically **adjust partitions**.

Understanding RDD Partitions in PySpark

RDD (Resilient Distributed Dataset) is the fundamental data structure in Apache Spark. It **divides large datasets into smaller partitions** and distributes them across multiple nodes in a cluster for **parallel processing**.

1. What is Partitioning in Spark?

Partitioning refers to how **Spark splits data into chunks (partitions) and distributes them across the cluster** for parallel computation.

- Each partition contains a **subset of the entire dataset**.
- Spark processes each partition **independently** in parallel.
- More partitions can improve performance **up to a certain point**.

2. Understanding `sc.parallelize(data, numPartitions)`

```
num = sc.parallelize([5,5,4,3,2,9,2,4,5,6,7,8,9], 9)
```

This command does the following:

- **Creates an RDD** from the given list `[5,5,4,3,2,9,2,4,5,6,7,8,9]`.
- **Divides the dataset into 9 partitions**, each containing approximately **one or two elements**.
- Spark distributes these partitions **across different nodes in the cluster**.

Example: Viewing Partitions

You can check how many partitions an RDD has using:

```
print(num.getNumPartitions())
# Output: 9
```

To see how data is distributed among partitions:

```
print(num.glom().collect())
```

This groups elements **by partitions** and returns:

```
[[5], [5], [4], [3], [2], [9], [2], [4], [5, 6, 7, 8, 9]]
```