01 SparkContext vs SparkSession - Nikhil X KirkYagami

Apache Spark: SparkContext vs SparkSession

1. Introduction to Spark Entry Points

What is an Entry Point?

An entry point in Apache Spark is the gateway to distributed computing capabilities. It serves as the connection between your application and the Spark cluster, managing resources and coordinating data processing operations.

Evolution Timeline

- **Pre-Spark 2.0 (2016)**: SparkContext was the primary entry point
- **Spark 2.0+**: SparkSession introduced as unified entry point
- **Current Best Practice**: SparkSession is preferred for most applications

Key Concepts

- **SparkContext**: Original, low-level entry point for RDD operations
- **SparkSession**: Modern, unified entry point supporting DataFrames, SQL, and streaming
- **Backward Compatibility**: SparkSession includes SparkContext internally

---

2. SparkContext Deep Dive

Overview

SparkContext is the foundational entry point in Spark, designed for low-level RDD (Resilient Distributed Dataset) operations. It provides fine-grained control over distributed data processing.

Key Responsibilities

1. **Cluster Connection**: Links application to cluster managers (Standalone, YARN, Kubernetes)
2. **Resource Management**: Allocates executors and manages memory/CPU
3. **RDD Operations**: Creates and manages RDDs with transformations and actions
4. **Configuration**: Sets application properties and cluster settings

Creating SparkContext in PySpark

```
from pyspark import SparkContext, SparkConf

# Method 1: Using SparkConf
conf = SparkConf()
conf.setAppName("MySparkApp")
conf.setMaster("local[*]")  # Use all available cores locally
conf.set("spark.executor.memory", "4g")
conf.set("spark.driver.memory", "2g")
```

```
sc = SparkContext(conf=conf)

# Method 2: Direct instantiation
sc = SparkContext(
    appName="MySparkApp",
    master="local[*]"
)
```

Important SparkConf Parameters

Parameter	Description	Example
`setAppName(name)`	Names the application	`"WordCount"`
`setMaster(url)`	Specifies cluster manager	`"local[*]"`, `"yarn"`, `"spark://host:port"`
`set(key, value)`	Custom properties	`"spark.executor.memory", "4g"`

Key SparkContext Methods

```
# Reading data
rdd = sc.textFile("path/to/file.txt", minPartitions=4)
rdd = sc.parallelize([1, 2, 3, 4, 5], numSlices=2)

# Configuration and control
sc.setLogLevel("WARN")  # Set logging level
config = sc.getConf()   # Get current configuration
sc.stop()               # Shutdown context
```

Complete Word Count Example with SparkContext

```
from pyspark import SparkContext, SparkConf

# Setup configuration
conf = SparkConf().setAppName("WordCount").setMaster("local[*]")
sc = SparkContext(conf=conf)

try:
    # Read input file
    lines = sc.textFile("input.txt")

    # Process data using RDD transformations
    words = lines.flatMap(lambda line: line.split(" "))
    word_pairs = words.map(lambda word: (word.lower().strip(), 1))
    word_counts = word_pairs.reduceByKey(lambda a, b: a + b)

    # Collect results
    results = word_counts.collect()

    # Display results
    for word, count in results:
        print(f"{word}: {count}")

    # Save results
    word_counts.saveAsTextFile("output_sparkcontext")
```

```
finally:
    sc.stop()
```

RDD Transformation Methods

Method	Description	Example
`map(func)`	Transform each element	`rdd.map(lambda x: x * 2)`
`flatMap(func)`	Transform and flatten	`rdd.flatMap(lambda x: x.split())`
`filter(func)`	Filter elements	`rdd.filter(lambda x: x > 0)`
`reduceByKey(func)`	Reduce values by key	`rdd.reduceByKey(lambda a, b: a + b)`

---

3. SparkSession Deep Dive

Overview

SparkSession is the unified entry point introduced in Spark 2.0, consolidating multiple contexts (SparkContext, SQLContext, StreamingContext) into a single API. It's the modern standard for Spark applications.

Key Responsibilities

1. **Unified Access**: Combines RDD, DataFrame, and SQL functionalities
2. **Cluster Connection**: Connects to cluster like SparkContext
3. **Resource Management**: Manages executors and configurations
4. **DataFrame/SQL Operations**: Supports structured data with optimizations
5. **Streaming & ML Support**: Enables real-time processing and machine learning

Creating SparkSession in PySpark

```
from pyspark.sql import SparkSession

# Basic creation
spark = SparkSession.builder \
    .appName("MySparkApp") \
    .master("local[*]") \
    .getOrCreate()

# Advanced configuration
spark = SparkSession.builder \
    .appName("AdvancedApp") \
    .master("local[*]") \
    .config("spark.executor.memory", "4g") \
    .config("spark.driver.memory", "2g") \
    .config("spark.sql.shuffle.partitions", "200") \
    .enableHiveSupport() \
    .getOrCreate()
```

SparkSession Builder Parameters

Method	Description	Example
`appName(name)`	Set application name	`.appName("MyApp")`
`master(url)`	Set cluster manager	`.master("local[*]")`
`config(key, value)`	Set configuration	`.config("spark.executor.memory", "4g")`
`getOrCreate()`	Get existing or create new session	Returns SparkSession instance

Key SparkSession Methods

```
# Data reading
df = spark.read.text("path/to/file.txt")
df = spark.read.csv("path/to/file.csv", header=True, inferSchema=True)
df = spark.read.json("path/to/file.json")

# SQL operations
spark.sql("SELECT * FROM table_name")

# DataFrame creation
df = spark.createDataFrame(data, schema=None)

# Access underlying SparkContext
sc = spark.sparkContext

# Configuration
spark.conf.set("spark.sql.shuffle.partitions", "100")

# Shutdown
spark.stop()
```

Complete Word Count Example with SparkSession

```
from pyspark.sql import SparkSession
from pyspark.sql.functions import explode, split, lower, trim

# Create SparkSession
spark = SparkSession.builder \
    .appName("WordCount") \
    .master("local[*]") \
    .getOrCreate()

try:
    # Read input file as DataFrame
    df = spark.read.text("input.txt")

    # Process using DataFrame operations
    words_df = df.select(
        explode(split(df.value, " ")).alias("word")
    ).select(
        lower(trim("word")).alias("word")
    ).filter(
        "word != ''"
    )
```

```
    # Count words
    word_counts = words_df.groupBy("word").count().orderBy("count", ascending=False)

    # Show results
    word_counts.show(20)

    # Save results
    word_counts.write.mode("overwrite").csv("output_sparksession")

finally:
    spark.stop()
```

Alternative SQL Approach

```
from pyspark.sql import SparkSession

spark = SparkSession.builder.appName("WordCountSQL").master("local[*]").getOrCreate()

try:
    # Read and create temporary view
    df = spark.read.text("input.txt")
    df.createOrReplaceTempView("lines")

    # Use SQL for word count
    result = spark.sql("""
        SELECT word, COUNT(*) as count
        FROM (
            SELECT explode(split(value, ' ')) as word
            FROM lines
        ) words
        WHERE word != ''
        GROUP BY word
        ORDER BY count DESC
    """)

    result.show()

finally:
    spark.stop()
```

---

4. Comprehensive Comparison

1. Purpose and Scope

Aspect	SparkContext	SparkSession
Primary Use	RDD-based processing	Unified API (RDD, DataFrame, SQL)
Data Focus	Unstructured data	Structured and semi-structured data
API Level	Low-level operations	High-level abstractions
Additional Contexts	Requires SQLContext, etc.	All-in-one solution

2. API Complexity

SparkContext Example:

```
# Verbose RDD operations
rdd = sc.textFile("file.txt")
result = rdd.flatMap(lambda x: x.split()) \
           .map(lambda x: (x, 1)) \
           .reduceByKey(lambda a, b: a + b)
```

SparkSession Example:

```
# Concise DataFrame operations
df = spark.read.text("file.txt")
result = df.select(explode(split("value", " ")).alias("word")) \
          .groupBy("word").count()
```

3. Performance Optimizations

Feature	SparkContext	SparkSession
Query Optimization	Manual optimization required	Automatic via Catalyst Optimizer
Memory Management	Basic RDD caching	Advanced Tungsten engine
Join Optimization	Manual implementation	Automatic broadcast joins
Predicate Pushdown	Not available	Automatic optimization

> Predicate Pushdown = pushing the filtering logic(like `WHERE` clauses) as close as to the data source so that less data is read into spark.

4. Feature Support Matrix

Feature	SparkContext	SparkSession
RDD Operations	Native	Via sparkContext
DataFrame API	`NA`, Requires SQLContext	Native
SQL Queries	`NA`, Requires SQLContext	Native
Structured Streaming	`NA`	Native
Machine Learning	MLlib (RDD-based)	ML (DataFrame-based)
Graph Processing	GraphX	GraphX via sparkContext

---

5. When to Use Which

Use SparkContext When:

1. **Legacy Code Maintenance**

```
# Maintaining pre-Spark 2.0 applications
sc = SparkContext(appName="LegacyApp")
```

2. **Low-level RDD Control**

```
# Custom partitioning or complex transformations
rdd = sc.parallelize(data, numSlices=custom_partitions)
custom_rdd = rdd.mapPartitions(custom_partition_function)
```

3. **Unstructured Data Processing**

```
# Processing raw binary data or complex text formats
binary_rdd = sc.binaryFiles("path/to/binary/files")
```

Use SparkSession When:

1. **Modern Applications (Recommended)**

```
# Default choice for new projects
spark = SparkSession.builder.appName("ModernApp").getOrCreate()
```

2. **Structured Data Processing**

```
# Working with CSV, JSON, Parquet files
df = spark.read.csv("data.csv", header=True, inferSchema=True)
```

3. **SQL Analytics**

```
# Running SQL queries
df.createOrReplaceTempView("sales")
result = spark.sql("SELECT * FROM sales WHERE amount > 1000")
```

4. **ETL Pipelines**

```
# Extract, Transform, Load operations
raw_df = spark.read.json("input.json")
transformed_df = raw_df.select("col1", "col2").filter("col1 > 0")
transformed_df.write.parquet("output.parquet")
```

---

6. Best Practices

1. Choosing the Right Entry Point

```
# Recommended: Use SparkSession as default
spark = SparkSession.builder \
    .appName("MyApp") \
    .master("local[*]") \
    .getOrCreate()

# Access SparkContext when needed for RDD operations
sc = spark.sparkContext
rdd = sc.parallelize([1, 2, 3, 4, 5])
```

2. Configuration Best Practices

```
# Good: Configure via SparkSession builder
spark = SparkSession.builder \
    .appName("OptimizedApp") \
    .config("spark.executor.memory", "4g") \
    .config("spark.executor.cores", "4") \
    .config("spark.sql.adaptive.enabled", "true") \
    .config("spark.sql.adaptive.coalescePartitions.enabled", "true") \
    .getOrCreate()

# Good: Runtime configuration changes
spark.conf.set("spark.sql.shuffle.partitions", "200")
```

3. Resource Management

```
# Always stop Spark context/session
try:
    # Your Spark operations
    df = spark.read.csv("data.csv")
    result = df.groupBy("category").count()
    result.show()
finally:
    spark.stop()  # This also stops underlying SparkContext
```

4. Combining Both APIs

```
# Best practice: Use SparkSession with SparkContext access
spark = SparkSession.builder.appName("HybridApp").getOrCreate()

try:
    # DataFrame operations
    df = spark.read.text("input.txt")

    # Convert to RDD when needed
    rdd = df.rdd.map(lambda row: row.value.upper())

    # Back to DataFrame
    result_df = spark.createDataFrame(rdd.map(lambda x: (x,)), ["upper_text"])

    result_df.show()
finally:
    spark.stop()
```

---

7. Practical Examples

Example 1: Log File Analysis

Using SparkContext:

```
from pyspark import SparkContext
```

```
sc = SparkContext(appName="LogAnalysis")

try:
    logs = sc.textFile("access.log")

    # Extract IP addresses
    ips = logs.map(lambda line: line.split()[0])

    # Count unique IPs
    ip_counts = ips.map(lambda ip: (ip, 1)) \
                   .reduceByKey(lambda a, b: a + b) \
                   .sortBy(lambda x: x[1], ascending=False)

    # Get top 10 IPs
    top_ips = ip_counts.take(10)

    for ip, count in top_ips:
        print(f"{ip}: {count}")

finally:
    sc.stop()
```

Using SparkSession:

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

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

try:
    # Read log file
    logs_df = spark.read.text("access.log")

    # Extract IP addresses using DataFrame operations
    ips_df = logs_df.select(
        split(col("value"), " ").getItem(0).alias("ip")
    )

    # Count and sort
    ip_counts = ips_df.groupBy("ip") \
                      .count() \
                      .orderBy("count", ascending=False)

    # Show top 10
    ip_counts.show(10)

finally:
    spark.stop()
```

Example 2: ETL Pipeline

```
from pyspark.sql import SparkSession
from pyspark.sql.functions import col, when, avg, max, min
from pyspark.sql.types import StructType, StructField, StringType, IntegerType, DoubleType

spark = SparkSession.builder \
```

```
    .appName("ETL_Pipeline") \
    .config("spark.sql.adaptive.enabled", "true") \
    .getOrCreate()

try:
    # Extract: Read from multiple sources
    sales_df = spark.read.csv("sales.csv", header=True, inferSchema=True)
    customers_df = spark.read.json("customers.json")

    # Transform: Clean and process data
    # Clean sales data
    clean_sales = sales_df.filter(col("amount") > 0) \
                          .withColumn("amount",
                                      when(col("amount") > 10000, 10000)
                                      .otherwise(col("amount")))

    # Join datasets
    enriched_data = clean_sales.join(customers_df, "customer_id", "left")

    # Aggregate data
    summary = enriched_data.groupBy("region") \
                           .agg(
                               avg("amount").alias("avg_amount"),
                               max("amount").alias("max_amount"),
                               min("amount").alias("min_amount")
                           )

    # Load: Save processed data
    summary.write.mode("overwrite").parquet("output/sales_summary")

    # Show results
    summary.show()

finally:
    spark.stop()
```

Summary

Key Takeaways

1. **SparkSession is the Modern Standard**: Use SparkSession for new applications as it provides a unified API with automatic optimizations.
2. **SparkContext for Legacy and Low-level Operations**: Use SparkContext only when maintaining legacy code or requiring fine-grained RDD control.
3. **Performance Advantage**: SparkSession leverages Catalyst Optimizer and Tungsten engine for better performance with structured data.
4. **Backward Compatibility**: SparkSession includes SparkContext, so you can access RDD operations when needed.
5. **Best Practice Pattern**:

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

# Access SparkContext when needed
sc = spark.sparkContext
```

```
# Always clean up
spark.stop()
```

Next Steps for Learning

1. **Explore DataFrame Operations**: Master the DataFrame API for structured data processing
2. **Learn Spark SQL**: Understand how to write efficient SQL queries in Spark
3. **Study Performance Tuning**: Learn about partitioning, caching, and optimization techniques
4. **Practice Streaming**: Implement real-time data processing with Structured Streaming
5. **Machine Learning**: Explore MLlib and ML pipelines for data science applications

Remember: SparkSession is your go-to choice for modern Spark applications, while SparkContext remains valuable for specific use cases requiring low-level control.