

✦ Get unlimited access to the best of Medium for less than \$1/week. [Become a member](#)



## Towards Data Engineering

# Understanding Spark Serialization and Deserialization: A Complete Guide



Sukrit Mehta

Follow

7 min read · Nov 18, 2024

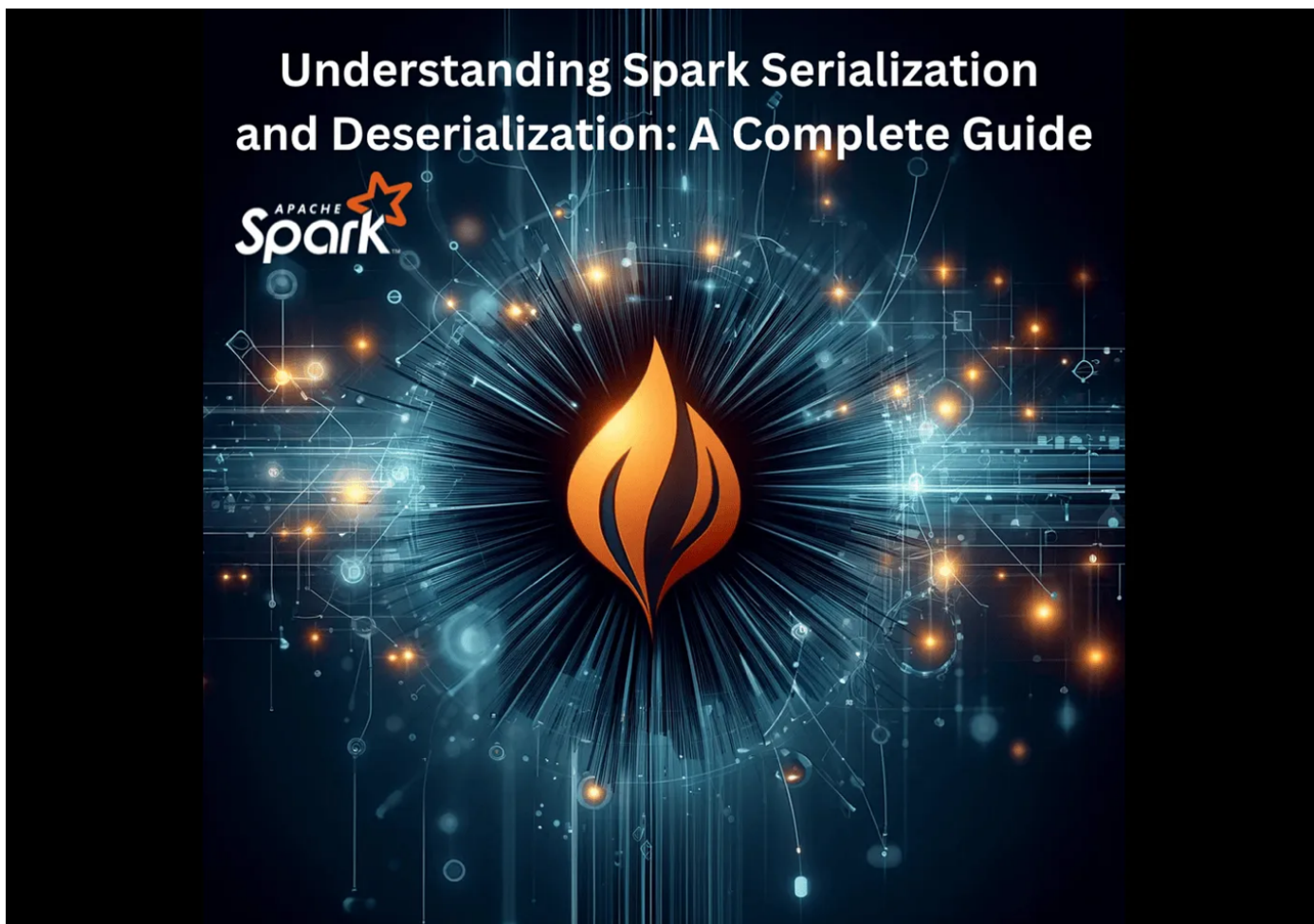


68



1





*Serialization* is a crucial concept in Apache Spark that affects the performance of data processing and transfers across distributed systems. In this guide, we'll walk through what serialization and deserialization are, why they matter in Spark, and how you can optimize them for faster, more efficient applications.

## What is Serialization and Deserialization?

- **Serialization:** The process of converting an object into a byte stream so that it can be transmitted over a network or stored efficiently.
- **Deserialization:** The reverse process — turning a byte stream back into an object.

*Looks complex ?*

## Let's understand with a real-world example for better understanding

Imagine you have a *handwritten letter* (maybe a love letter as well) that you want to send to a very good friend of yours who lives in another city. You can't physically deliver the letter in its original form (a piece of paper) because you both live far apart. Here's what you do instead:

1. **Serialization:** You put the letter into an envelope and address it. By doing this, you have transformed your letter into a format (the envelope) that is easy to transport using a postal service. The envelope is much simpler and more compact for the post office to handle.

2. **Transmission:** The post office then sends the envelope across the city to your friend's house.

3. **Deserialization:** When your friend receives the envelope, they *open it up* and take out the original letter to read it. This process is the reverse — converting the transported format (the envelope) back into something understandable (the letter).

### *How this relates to Serialization in Computing?*

In the world of computing:

- The *object* (like your letter) could be data structures, lists, or complex objects in a programming language.
- *Serialization* is the process of packaging the object into a byte stream

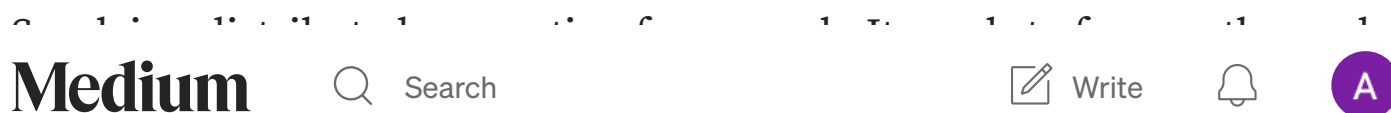
(like putting the letter into an envelope), making it suitable to send across a network or store in a file.

- **Deserialization** is when the receiving computer opens the byte stream and converts it back into the original object so it can be used again.

This concept ensures data can be efficiently transmitted or stored and then correctly reconstructed later.

Now let's dive deep into it.

## But why Does Spark Need Serialization ?



Serialization ensures these transfers are fast and memory-efficient.

### An Everyday Example :

Imagine you're working on a Spark job that applies transformations to a massive dataset. Whenever you use operations like *map* or *filter*, Spark needs to move data between nodes. Without efficient serialization, these data transfers would be slow and consume a lot of resources.

## How Does Spark Handle Serialization?

Spark offers two primary serialization libraries:

1. Java Serialization (default)
2. Kryo Serialization (recommended for better performance)

## 1. Java Serialization

Java's built-in serialization is the default for Spark. While easy to use, it isn't the most efficient.

- **How It Works:** Converts objects into byte streams using Java's *Serializable* interface.
- **Drawbacks:** It's relatively slow and can generate large serialized objects.

```
import org.apache.spark.sql.SparkSession
val spark = SparkSession.builder
  .appName("Java Serialization Example")
  .master("local[*]")
  .getOrCreate()

case class Person(name: String, age: Int)

val rdd = spark.sparkContext.parallelize(Seq(Person("Alice", 30), Person("Bob", 25)))

// Applying a transformation
val namesRDD = rdd.map(person => person.name)
namesRDD.collect().foreach(println)
```

## 2. Kryo Serialization

For better performance, use Kryo. It's faster and generates smaller serialized data. However, you need to register your classes.

(a) Setting Up Kryo Serialization:

```
import org.apache.spark.sql.SparkSession

val spark = SparkSession.builder
```

```
.appName("Kryo Serialization Example")
.master("local[*]")
.config("spark.serializer", "org.apache.spark.serializer.KryoSerializer")
.getOrCreate()
```

## (b) Registering Classes with Kryo:

```
import org.apache.spark.SparkConf

case class Person(name: String, age: Int, address: String)

val conf = new SparkConf()
conf.set("spark.serializer", "org.apache.spark.serializer.KryoSerializer")
conf.registerKryoClasses(Array(classOf[Person]))

val sparkContext = new org.apache.spark.SparkContext(conf)
```

## Comparison of Java and Kryo Serialization with code snippet runs:

### Common codebase to run :

```
import org.apache.spark.{SparkConf, SparkContext}
import org.apache.spark.sql.SparkSession
import scala.util.Random
import org.apache.spark.storage.StorageLevel

case class Person(name: String, age: Int, address: String)
// Function to generate random Person objects

def generateRandomPersons(n: Int): Seq[Person] = {
  val random = new Random()
  (1 to n).map { _ =>
    val name = s"Name_${random.alphanumeric.take(10).mkString}"
```

```
    val age = random.nextInt(100)
    val address = s"${random.nextInt(20000)} Random Street"
    Person(name, age, address)
  }
}
// Generate 20,000 random Person objects
val personData = generateRandomPersons(20000)
```

## Java Snippet

```
// Spark Configuration for Java Serialization
val confJava = new SparkConf().setAppName("Java Serialization Example").setM

// Spark Session using Java Serialization
val sparkJava = SparkSession.builder().config(confJava).getOrCreate()
val scJava = sparkJava.sparkContext

val rddJava = scJava.parallelize(personData)

rddJava.persist(StorageLevel.MEMORY_ONLY_SER)

// Perform an action to trigger serialization
rddJava.map(person => (person.name, person.age)).collect()
```

## Kryo Snippet

```
// Spark Configuration for Kryo Serialization
val confKryo = new SparkConf()
    .setAppName("Kryo Serialization Example")
    .setMaster("local[*]")
    .set("spark.serializer", "org.apache.spark.serializer.KryoSerializer")
    .registerKryoClasses(Array(classOf[Person])) // Register the Person class

// Spark Session using Kryo Serialization
val sparkKryo = SparkSession.builder()
    .config(confKryo)
```

```
.getOrCreate()

val scKryo = sparkKryo.sparkContext
// Create the same RDD of 100,000 Person objects

val rddKryo = scKryo.parallelize(personData)

// Perform the same action to trigger serialization
rddKryo.map(person => (person.name, person.age)).collect()
```

## Comparison of the outputs:

### Summary Metrics for 8 Completed Tasks

Metric	Min	25th percentile	Median	75th percentile	Max
Task Deserialization Time	0.3 s	0.3 s	0.3 s	0.3 s	0.3 s
Duration	1 s	1 s	1 s	1 s	1 s
GC Time	0.7 s	0.7 s	0.7 s	0.7 s	0.7 s
Result Serialization Time	0.0 ms	1.0 ms	1.0 ms	1.0 ms	17.0 ms
Getting Result Time	0.0 ms	0.0 ms	0.0 ms	0.0 ms	0.0 ms
Scheduler Delay	12.0 ms	20.0 ms	35.0 ms	0.1 s	0.1 s
Peak Execution Memory	0.0 B	0.0 B	0.0 B	0.0 B	0.0 B

### Summary metrics for the Java serialization

### Summary Metrics for 8 Completed Tasks

Metric	Min	25th percentile	Median	75th percentile	Max
Task Deserialization Time	0.3 s	0.3 s	0.3 s	0.3 s	0.3 s
Duration	9.0 ms	9.0 ms	10.0 ms	27.0 ms	29.0 ms
GC Time	14.0 ms	14.0 ms	14.0 ms	14.0 ms	21.0 ms
Result Serialization Time	8.0 ms	9.0 ms	15.0 ms	17.0 ms	25.0 ms
Getting Result Time	0.0 ms	0.0 ms	0.0 ms	0.0 ms	0.0 ms
Scheduler Delay	51.0 ms	67.0 ms	85.0 ms	97.0 ms	0.2 s
Peak Execution Memory	0.0 B	0.0 B	0.0 B	0.0 B	0.0 B

### Summary metrics for the Kryo serialization

As we can see that Kryo serialization takes less duration to complete in



comparison with Java serialization.

## When should you use Kryo?

- For large datasets where efficiency is key.
- When your application has **complex data structures** or custom objects.
- To reduce the memory footprint in memory-intensive jobs.

## Performance Boost: Java vs. Kryo

Switching from Java serialization to Kryo can:

- Make serialization *up to 10x faster*.
- Reduce memory usage significantly, especially for large or complex objects.

## Advanced Tips for Optimizing Serialization

### 1. *Avoid Unnecessary Serialization*

- Use *broadcast variables* for data that is shared across tasks, so it's only serialized once.

**Example:**

```
val broadcastVar = spark.sparkContext.broadcast(Array(1, 2, 3))
```

### 2. *Use Primitive Types When Possible*

- Primitives like Int, Long, and Double are more efficient than complex objects.

### 3. *Be Careful with Serializable*

— Avoid making large classes *Serializable* unless necessary, as it may lead to excessive overhead.

### 4. *Leverage the transient Keyword*

— Use *transient* for fields that don't need to be serialized.

— Example:

```
class ExampleClass extends Serializable {  
  @transient val tempData = new SomeLargeObject()  
}
```

## Important : Common Serialization Mistakes to Avoid

### 1. *Non-Serializable Objects*

— If an object isn't serializable, Spark will throw a *NotSerializableException*. Ensure your objects are serializable.

### 2. *Anonymous Functions Trap*

— Be mindful of capturing non-serializable variables within anonymous functions. This can lead to serialization errors.

### *Example of a Serialization Error:*

```
val nonSerializableObject = new SomeNonSerializableClass()  
val rdd = spark.sparkContext.parallelize(Seq(1, 2, 3))  
// This will throw a NotSerializableException  
rdd.map(x => nonSerializableObject.someMethod(x)).collect()
```

# When Kryo Serialization May Not Be Better Than Java Serialization

While Kryo is generally much faster and more memory-efficient than Java's built-in serialization, there are some specific cases where Kryo might not perform as well or might not be the ideal choice:

## 1. Simple or Small Objects

- If your data structures are **very simple** (like primitive types or small strings), the performance difference between Kryo and Java serialization might be negligible.
- **Overhead:** Setting up Kryo and registering classes can add complexity that isn't justified for lightweight objects. In these cases, the built-in Java serialization could suffice and simplify the implementation.

## 2. Serialization Setup and Complexity

- Kryo requires you to **register all custom classes** you want to serialize, which can be cumbersome in a large application. If your codebase is constantly changing or includes many different types of objects, this extra setup might outweigh the performance benefits.
- Forgetting to register classes with Kryo can lead to errors and unexpected behavior, making Java serialization a more convenient (though less efficient) option in scenarios where ease of use is a priority.

## 3. Compatibility Concerns

- **Backward and Forward Compatibility:** Kryo serialization does not guarantee backward or forward compatibility as well as Java serialization. If you are working on applications that require serialized objects to be stored and later read by different versions of your software, Java serialization may provide more reliability.
- For example, in systems where data is serialized to disk and may need to be deserialized years later (or by different versions of the application), Java serialization's robustness and compatibility features could be advantageous.

#### 4. Built-In Java Object Support

- Java serialization natively supports serializing a wide range of standard Java objects without any extra configuration. If you are dealing with **Java's built-in types**, you might not gain much from Kryo, and using Java serialization would simplify the implementation.
- Kryo, on the other hand, may require you to write custom serializers for certain complex or non-standard objects, adding to the development time and potential for errors.

## Conclusion

Serialization is a key factor that can make or break the performance of your Spark applications. By understanding and optimizing serialization, you can make your Spark jobs faster and more memory-efficient.

Remember:

- Use Kryo for performance-critical tasks.
- Optimize your data structures and serialization strategies.
- Always profile and test your Spark jobs to identify bottlenecks.

Mastering these techniques will help you take full advantage of Spark's distributed processing capabilities. Happy learning!

*Thank you for reading my article! If you found it helpful, a few claps 🙌 would mean a lot and inspire me to continue sharing valuable content. Feel free to follow me on [LinkedIn](#) and [Medium](#) to stay updated and connect for more insights!*

Data Engineering

Spark

Apache Spark Tutorial

Serialization



## Published in Towards Data Engineering

5.5K followers · Last published just now

[Follow](#)

Dive into data engineering with top Medium articles on big data, cloud, automation, and DevOps. Follow us for curated insights and contribute your expertise. Join our thriving community of professionals and enthusiasts shaping the future of data-driven solutions.



## Written by Sukrit Mehta

15 followers · 43 following

[Follow](#)

Big Data Engineer

## Responses (1)



Aaditya Adhikari

What are your thoughts?

---



deepak gupta

Nov 19, 2024



Great explanation sir! Would love to see more article like this.

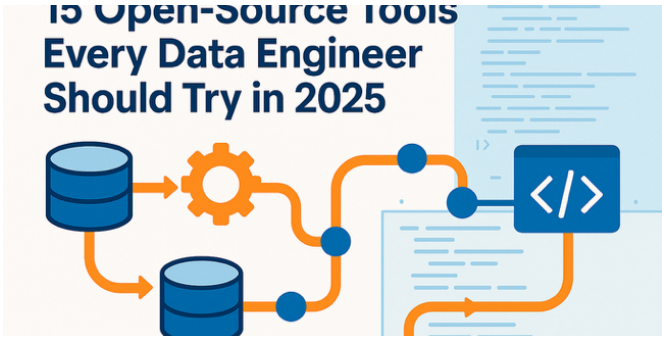


1

[Reply](#)

---

## More from Sukrit Mehta and Towards Data Engineering

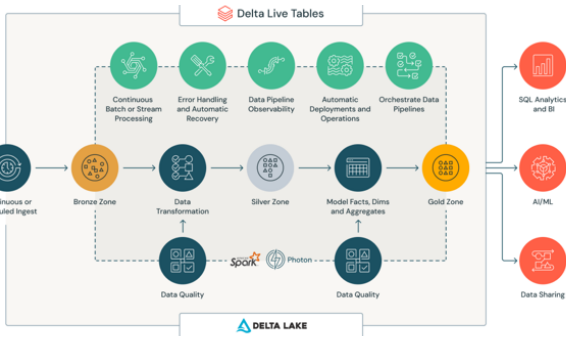


 In Towards Data Engine... by Victor Oketch Sa...

## 15 Open-Source Tools Every Data Engineer Should Try in 2025

How I leveled up my pipelines (and how you can too)

★ Jun 10 🤝 142 💬 2 📌 ⋮

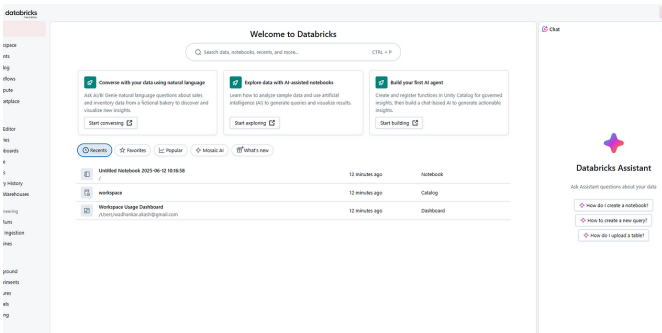


 In Towards Data Engine... by THE BRICK LEAR...

## Databricks Open Sources Declarative Pipelines: A New Er...

Databricks just made a landmark move in the data engineering ecosystem—it open...

★ 6d ago 🤝 55 📌 ⋮




 In Towards Data Engine... by THE BRICK LEAR...

## Databricks is Now Free for Learners—No Cloud Account...

In a major win for the data learning community, Databricks has just made its...

★ 6d ago 🤝 88 💬 3 📌 ⋮



 In Towards Data Engine... by THE BRICK LEAR...

## Databricks Data and AI Summit 2025 : Executive Summary

Databricks DAIS 2025 marked a transformative leap in unified data, AI, and...

★ 5d ago 🤝 4 📌 ⋮

See all from Sukrit Mehta

See all from Towards Data Engineering

## Recommended from Medium

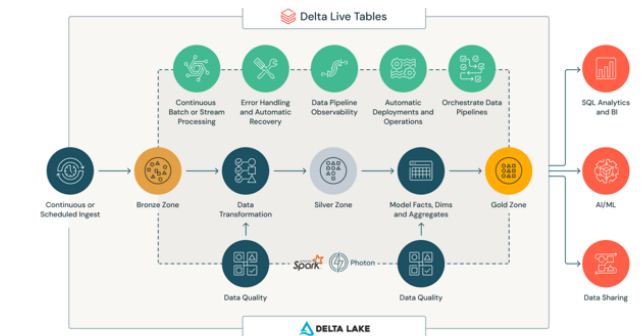


 Sriw World of Coding

### Processing 10 TB Data in Apache Spark in 10 Minutes: Ho...

 Introduction

May 6  32  1



 In Towards Data Engine... by THE BRICK LEAR...

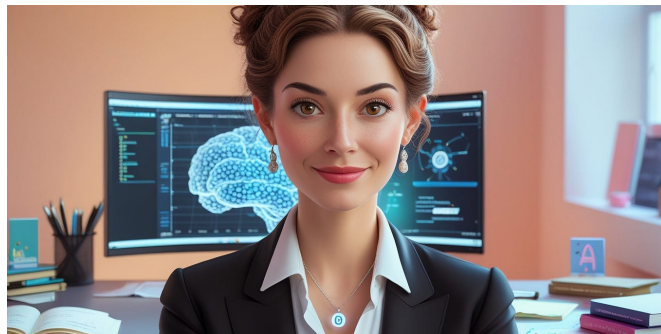
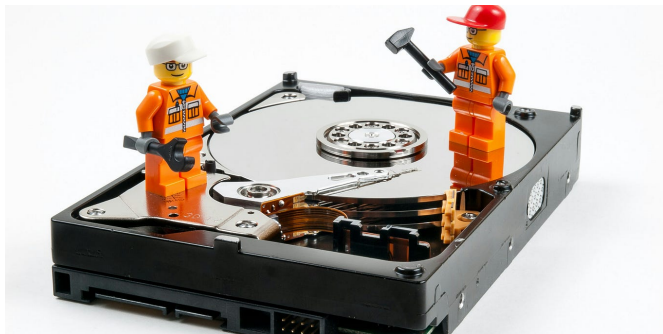
### Databricks Open Sources Declarative Pipelines: A New Er...

Databricks just made a landmark move in the data engineering ecosystem—it open...

★ 6d ago  55







In Data Engineer Things by Santosh Joshi

## Caching and Persisting in PySpark

Explore Differences Between Caching and Persisting in PySpark



Dec 30, 2024



67



In Python in Plain Engli... by Mayurkumar Sura...

## Mastering PySpark: 15 Production-Grade Interview...

The brutal truth about PySpark interviews and how to ace them with real-world...



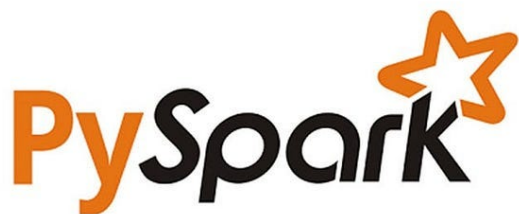
Jun 10



69



1



Mukovhe Mukwevho

## 5 Ways to Check If a Spark DataFrame is Empty

Read here for free if you do not have a medium subscription!



Feb 13



43



1



Praveen Kumar B N

## Apache Spark 4.0 : What really matters to Data Engineers

Intro



6d ago



10



See more recommendations

---

[Help](#) [Status](#) [About](#) [Careers](#) [Press](#) [Blog](#) [Privacy](#) [Rules](#) [Terms](#) [Text to speech](#)