**What is Apache Spark?**

Apache Spark is an open source, fast, general-purpose, in-memory processing engine for big data processing. Apache Spark was developed in 2009 at the University of California Berkeley’s AMP Lab and later open sourced as an Apache project in 2010. Apache Spark is written in Scala and provides high-level application programming interfaces (APIs) in Java, Scala, Python, and R.

**Why Apache Spark?**

Apache Spark provides a unified framework to perform different tasks that would have previously required different engines for processing such as batch, real-time processing.

• Apache Spark provides high-level operators (e.g., map, filter, etc.) to process the data that are not available in Hadoop MapReduce.

• Apache Spark is 100 times faster than Hadoop MapReduce when you run Spark in memory and 10 times faster than Hadoop MapReduce even when you run Spark on disk.

• Apache Spark supports both batch processing and real-time processing.

• Apache Spark provides an interactive shell that you can use for learning and exploring data.

• Apache Spark is not bundled with a storage system. Local file systems, Hadoop Distributed File System (HDFS), Cassandra, S3, and others can be used as storage systems.

**Spark Vs Hadoop MapReduce**

**The key difference between Hadoop MapReduce and Spark**

To make the comparison fair, here we will contrast Spark with Hadoop MapReduce, as both are responsible for data processing. In fact, the key difference between them lies in the approach to processing: Spark can do it in-memory, while Hadoop MapReduce has to read from and write to a disk. As a result, the speed of processing differs significantly – Spark may be up to 100 times faster. However, the volume of data processed also differs: Hadoop MapReduce is able to work with far larger data sets than Spark.



**Tasks Hadoop MapReduce is good for:**

Linear processing of huge data sets. Hadoop MapReduce allows parallel processing of huge amounts of data. It breaks a large chunk into smaller ones to be processed separately on different data nodes and automatically gathers the results across the multiple nodes to return a single result. In case the resulting dataset is larger than available RAM, Hadoop MapReduce may outperform Spark.

Economical solution, if no immediate results are expected. Our [Hadoop team](https://www.scnsoft.com/services/big-data/hadoop) considers MapReduce a good solution if the speed of processing is not critical. For instance, if data processing can be done during night hours, it makes sense to consider using Hadoop MapReduce.

**Tasks Spark is good for:**

Fast data processing. In-memory processing makes Spark faster than Hadoop MapReduce – up to 100 times for data in RAM and up to 10 times for data in storage.

Iterative processing. If the task is to process data again and again – Spark defeats Hadoop MapReduce. Spark’s Resilient Distributed Datasets (RDDs) enable multiple map operations in memory, while Hadoop MapReduce has to write interim results to a disk.

Near real-time processing. If a business needs immediate insights, then they should opt for Spark and its in-memory processing.

Graph processing. Spark’s computational model is good for iterative computations that are typical in graph processing. And Apache Spark has GraphX – an API for graph computation.

Machine learning. Spark has MLlib – a built-in machine learning library, while Hadoop needs a third-party to provide it. MLlib has out-of-the-box algorithms that also run in memory.

Joining datasets. Due to its speed, Spark can create all combinations faster, though Hadoop may be better if joining of very large data sets that requires a lot of shuffling and sorting is needed.

**What Sets Spark Apart?**

There are multiple reasons to choose Apache Spark, out of which the most significant ones are given below:

Speed: For large-scale processing of data, Spark is 100 times faster than Hadoop, regardless of the fact that data is stored in memory or on disk. Even if the data is stored on disk, Spark will be performing faster. Spark has a world record in on-disk sorting for large-scale data.

Ease of use: Spark has a crystal-clear and declarative approach toward a cluster of datasets. It has a collection of operators for data transformation, APIs specific to the dataset domain, or dataframes to manipulate semi-structured and structured data. Spark also has a single-entry point for applications.

Simplicity: Spark is designed in such a way that it can be easily accessible just by rich APIs. It is specially designed for quick and easy interaction in large data scale. APIs are well-documented for application developers and Data Scientists to instantly start working on Spark.

Support: As mentioned earlier, Spark supports too many programming languages like Python, Scala, Java, R, etc. It also integrates with other storage solutions based on Hadoop ecosystem, such as MapR, Apache Cassandra, Apache HBase, and Apache Hadoop (HDFS).

Spark Clusters and the Resource Management System

Spark is essentially a distributed system that was designed to process a large volume of data efficiently and quickly. This distributed system is typically deployed onto a collection of machines, which is known as a Spark *cluster*. A cluster size can be as small as a few machines or as large as thousands of machines. To efficiently and intelligently manage a collection of machines, companies rely on a resource management system such as Apache YARN or Apache Mesos. The two main components in a typical resource management system are the *cluster manager* and the *worker*. The cluster manager knows where the workers are located, how much memory they have, and the number of CPU cores each one has. One of the main responsibilities of the cluster manager is to orchestrate the work by assigning it to each worker. Each worker offers resources (memory, CPU, etc.) to the cluster manager and performs the assigned work.



Spark Driver

A Spark application consists of two parts. The first is the application data processing logic expressed using Spark APIs, and the other is the Spark driver. The application data processing logic can be as simple as a few lines of code to perform a few data processing operations or can be as complex as training a large machine learning model that requires many iterations and could run for many hours to complete. The Spark driver is the central coordinator of a Spark application, and it interacts with a cluster manager to figure out which machines to run the data processing logic on. For each one of those machines, the Spark driver requests that the cluster manager launch a process called the Spark executor. Another important job of the Spark driver is to manage and distribute Spark tasks onto each executor on behalf of the application. If the data processing logic requires the Spark driver to display the computed results to a user, then it will coordinate with each Spark executor to collect the computed result and merge them together. The entry point into a Spark application is through a class called SparkSession, which provides facilities for setting up configurations as well as APIs for expressing data processing logic.

Spark Executor

Each Spark executor is a JVM process and is exclusively allocated to a specific Spark application. This was a conscious design decision to avoid sharing a Spark executor between multiple Spark applications in order to isolate them from each other so one badly behaving Spark application wouldn’t affect other Spark applications. The lifetime of a Spark executor is the duration of a Spark application, which could run for a few minutes or for a few days. Since Spark applications are running in separate Spark executors, sharing

data between them will require writing the data to an external storage system like HDFS. Spark employs a master-slave architecture, where the Spark driver is the master and the Spark executor is the slave. Each of these components runs as an independent process on a Spark cluster. A Spark application consists of one

and only one Spark driver and one or more Spark executors. Playing the slave role, each Spark executor does what it is told, which is to execute the data processing logic in the form of tasks. Each task is executed on a separate CPU core. This is how Spark can speed up the processing of a large amount of data by processing it in parallel. In addition to executing assigned tasks, each Spark executor has the responsibility of caching a portion of the data in memory and/or on disk when it is told to do so by the application logic.



Spark Unified Stack

Spark provides a unified data processing engine known as the Spark stack. Similar to other well-designed systems, this stack is built on top of a strong foundation called Spark Core, which provides all the necessary functionalities to manage and run distributed applications such as scheduling, coordination, and fault tolerance. In addition, it provides a powerful and generic programming abstraction for data

processing called resilient distributed datasets (RDDs). On top of this strong foundation is a collection of components where each one is designed for a specific data processing workload, as shown in figure. Spark SQL is for batch as well as interactive data processing. Spark Streaming is for real-time stream data processing. Spark GraphX is for graph processing. Spark MLlib is for machine learning. Spark R is for running machine learning tasks using the R shell.



This unified engine brings several important benefits to the task of building scalable and intelligent big data applications. First, applications are simpler to develop and deploy because they use a unified set of APIs and run on a single engine. Second, it is way more efficient to combine different types of data processing (batch, streaming, etc.) because Spark can run those different sets of APIs over the same data without writing the intermediate data out to a storage system. Finally, the most exciting benefit is Spark enables new applications that were not possible before because of its ease of composing different sets of data processing types within a Spark application. For example, it can run interactive queries over the results of machine learning predictions of real-time data streams.

Spark Core

Spark core provides all the necessary functionalities to manage and run distributed applications such as scheduling, coordination, and fault tolerance. It is the bedrock of the Spark distributed data processing engine. It consists of two parts: the distributed computing infrastructure and the RDD programming abstraction. The distributed computing infrastructure is responsible for the distribution, coordination, and scheduling of computing tasks across many machines in the cluster. This enables the ability to perform parallel data processing of a large volume of data efficiently and quickly on a large cluster of machines. Two other important responsibilities of the distributed computing infrastructure are handling of computing task failures and efficiently moving data across machines, which is known as *data shuffling*. Advanced users of Spark need to have intimate knowledge of the Spark distributed computing infrastructure to be effective at designing highly performant Spark applications. RDDs are immutable, fault-tolerant, parallel data structures that let users explicitly persist intermediate results in memory, control their partitioning to optimize data placement, and manipulate them using rich set of operators. Essentially, it provides a set of APIs for Spark application developers to easily and efficiently perform large-scale data processing without worrying where data resides on the cluster or dealing with machine failures. The RDD APIs are exposed in multiple programming languages (Scala, Java, and Python), and they allow users to pass local functions to run on the cluster, which is something that is powerful and unique.

Resilient Distributed Dataset (RDD)

RDDs represent both the idea of how a large dataset is represented in Spark and the abstraction for working with it. RDDs are immutable, fault-tolerant, parallel data structures that let users explicitly persist intermediate results in memory, control their partitioning to optimize data placement, and manipulate them using a rich set of operators.

Immutable

RDDs are designed to be immutable, which means you can’t specifically modify a particular row in the dataset represented by that RDD. You can call one of the available RDD operations to manipulate the rows in the RDD into the way you want, but that operation will return a new RDD. The basic RDD will stay unchanged, and the new RDD will contain the data in the way that you want. The immutability of RDDs essentially requires an RDD to carry its lineage information that Spark leverages to efficiently provide the fault tolerance capability.

Fault Tolerant

The ability to process multiple datasets in parallel usually requires a cluster of machines to host and execute the computational logic. If one or more of those machines dies or becomes extremely slow because of unexpected circumstances, then how will that affect the overall data processing of those datasets? The good news is that Spark automatically takes care of handling the failure on behalf of its users by rebuilding the failed portion using the lineage information.

Parallel Data Structures

Imagine the use case where someone gives you a large log file that is 1TB size and you are asked to find out how many log statements contain the word exception in it. A slow solution would be to iterate through that log file from the beginning to the end and execute the logic of determining whether a particular log statement contains the word exception. A faster solution would be to divide that 1TB file into several chunks and execute the aforementioned logic on each chunk in a parallelized manner to speed up the overall processing time. Each chunk contains a collection of rows. The collection of rows is essentially the data structure that holds a set of rows and provides the ability to iterate through each row. Each chunk contains a collection of rows, and all the chunks are being processed in parallel. This is where the phrase parallel data structures comes from.

In-memory computation

RDD pushes the speed boundary (computation speed) by introducing a novel idea, which is the ability to do distributed in-memory computation. Machine learning algorithms are iterative in nature, meaning they need to go through many iterations to arrive at an optimal state. This is where distributed in-memory computation can help in reducing the completion time from days to hours. Another use case that can hugely benefit from distributed in-memory computation is interactive data mining, where multiple ad hoc queries are performed on the same subset of data. If that subset of data is persisted in memory, those queries will take seconds and not minutes to complete.

Rich set of Operations

RDDs provide a rich set of commonly needed data processing operations. They include the ability to perform data transformation, filtering, grouping, joining, aggregation, sorting, and counting. One thing to note about these operations is that they operate at the coarse-grained level, meaning the same operation is applied to many rows, not to any specific row. In summary, an RDD is represented as an abstraction and is defined by the following five pieces of information:

• A set of partitions, which are the chunks that make up the entire dataset

• A set of dependencies on parent RDDs

• A function for computing all the rows in the data set

• Metadata about the partitioning scheme (optional)

• Where the data lives on the cluster (optional); if the data lives on HDFS, then it would be where the block locations are located

The RDD operations are classified into two types: transformations and actions.

Transformations and Actions

In Spark, the core data structures are immutable, meaning they cannot be changed after they’re created. This might seem like a strange concept at first: if you cannot change it, how are you supposed

to use it? To “change” a DataFrame, you need to instruct Spark how you would like to modify it to do

what you want. These instructions are called transformations.

Transformation operations are lazily evaluated, meaning Spark will delay the evaluations of the invoked operations until an action is taken. In other words, the transformation operations merely record the specified transformation logic and will apply them at a later point. On the other hand, invoking an action operation will trigger the evaluation of all the transformations that preceded it, and it will either return some result to the driver or write data to a storage system, such as HDFS or the local file system.

**Lazy Evaluation**

Lazy evaluation means that Spark will wait until the very last moment to execute the graph of computation instructions. In Spark, instead of modifying the data immediately when you express some operation, you build up a plan of transformations that you would like to apply to your source data. By waiting until the last minute to execute the code, Spark compiles this plan from your raw DataFrame transformations to a streamlined physical plan that will run as efficiently as possible across the cluster. This provides immense benefits because Spark can optimize the entire data flow from end to end. An example of this is something called predicate pushdown on DataFrames. If we build a large Spark job but specify a filter at the end that only requires us to fetch one row from our source data, the most efficient way to execute this is to access the single record that we need. Spark will actually optimize this for us by pushing the filter down automatically.

Creating RDDs

There are three ways to create an RDD.

The first way to create an RDD is to parallelize an object collection, meaning converting it to a distributed dataset that can be operated in parallel.

firstRdd = spark.sparkContext.parallelize([‘Hello world’, ‘Welcome to the world of Spark Programming’], 2)

The second way to create an RDD is to read a dataset from a storage system, which can be a local computer file system, HDFS, Cassandra, Amazon S3, and so on.

secondRdd = spark.sparkContext.textFile(‘file ‘)

The third way to create an RDD is by invoking one of the transformation operations on an existing RDD.

thirdRdd = firstRdd.flatMap( lambda word: word.split(‘ ‘))

**Overview of Structured API Execution**

To execute code, we must write code. This code is then submitted to Spark either through the console or via a submitted job. This code then passes through the Catalyst Optimizer, which decides how the code should be executed and lays out a plan for doing so before, finally, the code is run and the result is returned to the user.



**Logical Planning**

The first phase of execution is meant to take user code and convert it into a logical plan. This logical plan only represents a set of abstract transformations that do not refer to executors or drivers, it’s purely to convert the user’s set of expressions into the most optimized version. It does this by converting user code into an unresolved logical plan. This plan is unresolved because although your code might be valid, the tables or columns that it refers to might or might not exist. Spark uses the catalog, a repository of all table and DataFrame information, to resolve columns and tables in the analyzer. The analyzer might reject the unresolved logical plan if the required table or column name does not exist in the catalog. If the analyzer can resolve it, the result is passed through the Catalyst Optimizer, a collection of rules that attempt to optimize the logical plan by pushing down predicates or selections. Packages can extend the Catalyst to include their own rules for domain specific optimizations.



**Physical Planning**

After successfully creating an optimized logical plan, Spark then begins the physical planning process. The physical plan, often called a Spark plan, specifies how the logical plan will execute on the cluster by generating different physical execution strategies and comparing them through a cost model.

