# Spark

## A general-purpose big data framework

*Paul Krzyzanowski*
*March 24, 2021*

> ***Goal:** Create a general-purpose, fault-tolerant distributed framework for analyzing large-scale data and data streams.*

## Introduction

MapReduce turned out to be an incredibly useful and widely-deployed framework for processing large amounts of data. However, its design forces programs to comply with its computation model, which is:

1. *Map*: create a sequence of key, value data.
2. *Shuffle-sort*: combine common keys together and partition them to reduce workers.
3. *Reduce*: process each unique key with all of its associated values.

It turned out that many applications had to run MapReduce over multiple passes to process their data. All intermediate data had to be stored back in the file system (GFS at Google, HDFS elsewhere), which tended to be slow since stored data was not just written to disks but also replicated. Moreover, the next MapReduce phase could not start until the previous MapReduce job completed fully.

MapReduce was also restricted in where it could read its data. It would generally be in GFS/HDFS files or some some storage structure built on top of them. In many cases, however, data resides within an SQL database or is streaming from other services (e.g, activity logs, remote monitoring).

**Spark** is a framework that provides a highly flexible and general-purpose way of dealing with big data processing needs, does not impose a rigid computation model, and supports a variety of input types. This enables Spark to deal with text files, graph data, database queries, and streaming sources and not be confined to a two-stage processing model. Programmers can develop arbitrarily-complex, multi-step data pipelines arranged in an arbitrary directed acyclic graph (DAG) pattern.

Programming in Spark involves defining a sequence of *transformations* and *actions*. Spark has support for a *map* action and a *reduce* operation, so it can implement traditional MapReduce operations but it also supports SQL queries, graph processing, and machine learning. Unlike MapReduce, Spark stores its intermediate results in memory, providing for dramatically higher performance.

## Spark architecture: tasks

An application that uses Spark identifies data sources and the operations on that data. The main application, called the **driver program** is linked with the Spark API, which creates a **SparkContext**. This is the heart of the Spark system and coordinates all processing activity. This SparkContext in the driver program connects to a Spark **cluster manager**. The cluster manager responsible for allocating worker nodes, launching **executors** on them, and keeping track of their status.

Each **worker node** runs one or more **executors**. An executor is a process that runs an instance of a Java Virtual Machine (JVM). When each executor is launched by the manager, it establishes a connection back to the driver program. The executor runs **tasks** on behalf of a specific SparkContext (application) and keeps related data in memory or disk storage. A task is a transformation or action. The executor remains running for the duration of the application. This provides a performance advantage over the MapReduce approach since new tasks can be started very quickly.

The executor also maintains a cache, which stores recently-used and frequently-used data in memory instead of having to store it to a disk-based file as the MapReduce framework does.

The driver goes through the user's program, which consists of actions and transformations on data and converts that into a series of **tasks**. The driver then sends these tasks to the executors that registered with it. A **task** is application code that runs in the executor on a Java Virtual Machine (JVM) and can be written in languages such as Scala, Java, Python, Clojure, and R. It is transmitted as a jar file to an executor, which then runs it.

## Spark architecture: Resilient Distributed Datasets (RDD)

Data in Spark is a collection of **Resilient Distributed Datasets** (**RDD**s). This is often a huge collection of stuff. Think of an individual RDD as a giant table in a database or a structured file.

Data is organized into RDDs. An RDD will be partitioned (sharded) across many computers so each task will work on only a part of the dataset (divide and conquer!). RDDs can be created in three ways:

1. They can be present as any file stored in HDFS or any other storage system supported in Hadoop. This includes Amazon S3 (a key-value server, similar in design to Dynamo), HBase (Hadoop's version of Bigtable), and Cassandra (a no-SQL eventually-consistent database). This data is created by other services, such as event streams, text logs, or a database. For instance, the results of a specific query can be treated as an RDD. A list of files in a specific directory can also be an RDD.

2. RDDs can be streaming sources using the Spark Streaming extension. This could be a stream of events from remote sensors, for example. For fault tolerance, a *sliding window* is used, where the contents of the stream are buffered in memory for a predefined time interval.

3. An RDD can be the output of a *transformation* function. This allows one task to create data that can be consumed by another task and is the way tasks pass data around. For example, one task can filter out unwanted data and generate a set of key-value pairs, writing them to an RDD. This RDD will be cached in memory (overflowing to disk if needed) and will be read by a task that reads the output of the task that created the key-

value data.

RDDs have specific properties:

- They are **immutable**. That means their contents cannot be changed. A task can read from an RDD and create a new RDD but it cannot modify an RDD. The framework magically garbage collects unneeded intermediate RDDs.
- They are typed. An RDD will have some kind of structure within in, such as a key-value pair or a set of fields. Tasks need to be able to parse RDD streams.
- They are partitioned. Spark breaks an RDD into smaller chunks (partitions) and spreads them across the available workers. Each partition is run as a separate concurrent task. A multi-processor or multi-core system shoudl get multiple partitions to make use of all cores. By default, Spark creates a partition for each block of an input file. The default size for HDFS is 128MB. Spark operations that generate RDDs may specify a partitioning function to shuffle the output data. THe default partitioning function is to send each generated row of data to the worker corresponding to *hash(key) mod workercount*.

They also have an optional property:

- They may be ordered. An RDD will often be a set of elements that can be sorted. In the case of key-value lists, the elements will be sorted by a key. The sorting function can be defined by the programmer but sorting enables one to implement things like *Reduce* operations.

## RDD operations

Spark allows two types of operations on RDDs: **transformations** and **actions**. **Transformations** read an RDD and return a new RDD. Example transformations are *map*, *filter*, *groupByKey*, and *reduceByKey*. Transformations are evaluated lazily, which means they are computed only when some task wants their data (the RDD that they generate). At that point, the driver schedules them for execution.

**Actions** are operations that evaluate and return a new value. When an action is requested on an RDD object, the necessary transformations are computed and the result is returned. Actions tend to be the things that generate the final output needed by a program. Example actions are *reduce*, *grab samples*, and *write to file*.

### Sample transformations

| Transformation | Description |
|---|---|
| map(func) | Pass each element through a function func |
| filter(func) | Select elements of the source on which func returns true |
| flatmap(func) | Each input item can be mapped to 0 or more output items |
| sample(withReplacement, fraction, seed) | Sample a fraction fraction of the data, with or without replacement, using a given random number generator seed |

| Transformation | Description |
|---|---|
| union(otherdataset) | Union of the elements in the source data set and otherdataset |
| distinct([numtasks]) | The distinct elements of the source dataset |
| groupByKey([numtasks]) | When called on a dataset of (K, V) pairs, returns a dataset of (K, seq[V]) pairs |
| reduceByKey(func, [numtasks]) | Aggregate the values for each key using the given reduce function |
| sortByKey([ascending], [numtasks]) | Sort keys in ascending or descending order |
| join(otherDataset, [numtasks]) | Combines two datasets, (K, V) and (K, W) into (K, (V, W)) |
| cogroup(otherDataset, [numtasks]) | Given (K, V) and (K, W), returns (K, Seq[V], Seq[W]) |
| cartesian(otherDataset) | For two datasets of types T and U, returns a dataset of (T, U) pairs |

### Sample actions

| Action | Description |
|---|---|
| reduce(func) | Aggregate elements of the dataset using func. |
| collect(func, [numtasks]) | Return all elements of the dataset as an array |
| count() | Return the number of elements in the dataset |
| first() | Return the first element of the dataset |
| take(n) | Return an array with the first n elements of the dataset |
| takeSample(withReplacement, fraction, seed) | Return an array with a random sample of num elements of the dataset |

## Data storage

Spark does not care how data is stored. The appropriate RDD connector determines how to read data. For example, RDDs can be the result of a query in a Cassandra database and new RDDs can be written to Cassandra tables. Alternatively, RDDs can be read from HDFS files or written to an HBASE table.

## Fault tolerance

For each RDD, the driver tracks the sequence of transformations used to create it. That means every RDD knows which task needed to create it. If any RDD is lost (e.g., a task that creates one died), the driver can ask the task that generated it to recreate it. The driver maintains the entire dependency graph, so this recreation may end up being a chain of transformation tasks going back to the original data.

## Sample program

Here is a sample of a trivially small program that uses Spark and processes log files.

The input data comes from some file sitting in HDFS (let's assume it's a huge one). We specify the use of a "textFile" connector to read the data.

```
// base RDD
val lines = sc.textFile("hdfs://some_file.log")
// transformed RDDs
val errors = lines.filter(_.startsWith("ERROR"))
val messages = errors.map(_.split("\t")).map(r => r(1))
messages.cache()

// action 1
messages.filter(_.contains("mysql")).count()
// action 2
messages.filter(_.contains("php")).count()
```

The first transformation is a *filter* that picks out lines that start with the text "ERROR". It creates a new RDD that we will call `errors`. This contains only the lines with error messages. Anything that is not an error is ignored. The next transformation (*map*) takes those error messages and splits each string by tabs. The output of that RDD is fed into another *map* transformation that extracts element 1, the token that immediately follows the ERROR string. In this example, the token is a string that identifies the program that created the error. The **cache** directive that follows, instructs the framework to cache the `messages` RDD. We will have two transformations that read from the `messages` RDD, so the *cache* directive is a hint to the framework to hold onto that RDD in memory.

We apply a **filter** transformation to extract only the lines from the `messages` RDD that contain the string `mysql`. Then we apply the **count** action to count the number of lines that are in the resulting dataset. This count tells us the number of errors that resulted from mysql.

In the last line, we again apply a **filter** transformation but extract only the lines that contain `php`. Then we count the number of lines in the resulting dataset with the **count** action. This tells us the number of errors due to php.

## Beyond RDDs

The Resilient Distributed Dataset (RDD) is the core component of Spark. As Spark evolved, two additional datasets were added to the framework: *Datasets* and *DataFrames*. Both of these were added to support structured data (columns of data) and are particularly useuful for making SQL queries via Spark.

An RDD is an arbitrary collection of data. The Spark framework is unaware of its structure and it is up to the program to parse its contents.

A **DataFrame** is a distributed collection of data organized into named colums. It is a table: a two-dimensional structure of columns and rows. This is comparable to a table in a relational database. DataFrames introduce a schema that goes along with the data and describes the column-based structure. Along with DataFrames, the Spark framework gained a *catalyst* *optimizer*. This allows Spark to optimize certain operations on it since it knows the data types and does not have to rely on using Java object serialization. DataFrames are particulalry useful for building relational queries. For example, you can filter data: `df.filter("age < 18");`. A programmer can convert a DataFrame to an RDD with an `rdd` method and convert an RDD to a DataFrame with a `toDF` method.

A **Dataset** API builds on top of DataFrames and provides a type-safe, object-oriented programming interface. It introduces *encoders* that translate between JVM objects and Spark's internal fomat. These allow access to indivudual attributes without the need to de-serialize an entire object. Datasets provide the best of DataFrames and RDDs. They provide the type-safe and functional-programming interface of RDDs along with the relational column-based model of DataFrames.

## Spark ecosystem

Spark has rich and growing ecosystem of supporting services:

- **Spark Streaming** is designed to enable Spark to process real-time streaming data. It uses a technique called **micro-batch** processing, where incoming data is accumulated into small chunks and those chunks are then processed.
- **Spark SQL** provides access to Spark data over a JDBC interface and converts SQL-like queries into Spark jobs. It allows people to process data using database queries rather than writing Spark transformations and actions.
- **Spark Mlib** is a machine learning library. It provides various utilities for classification, regression, clustering, and filtering of data. Programmers can use it to create machine learning pipelines, evaluate models, and tune hyper-parameters.
- **Spark GraphX** is designed to support graph computation. It adds a Pregel API to Spark and extends RDDs by introducing a directed multi-graph with properties attached to each vertex & edge. GraphX provides a set of graph-specific operations to create subgraphs, join vertices, aggregate messages, and so on.

### Spark Streaming

Let's take a closer look at how Spark Streaming supports continious data streams.

Big data systems such as MapReduce, BSP/Hama, and Pregel/Giraph expect to work on static data. The files are complete and unchanging before any processing on them can begin. Spark does the same. Some data sources, however, don't have a start or an end. These include data feeds that might be used for developing fraud models for bank transactions, generating statistics on the movement of people or cars, or collecting error data from machines in a factory.

Spark Streaming is an extension that allows Spark to process live data streams. It uses the same programming interface and operations as Spark does for static data. The difference is the use of **micro-batching**. With micro-batching, incoming data is collected into chunks that represent a time interval. The programmer can define that time interval. This chunking of data creates what Spark calls a **discretized stream**, or **DStream**.

A DStream is a continuous sequence of RDDs. Each of the RDDs represents a micro batch – data that was collected over an interval of time. DStreams are treated just like regular RDDs by the framework. You can apply transformations and actions on them as on any other RDDs. Every operation on a DStream is translated to an operation on a sequence of RDDs.

The distinction is that we have a never-ending stream of RDDs and the programmer needs to be aware of that.

### Spark summary

Spark is widely used and very well supported. The framework is fast. In many cases it can be 10x faster if using disk and 100x faster if caching results in memory than comparable MapReduce jobs. Storing intermediate results in memory and allowing multiple transformations or actions read from the same dataset are both keys to this high performance.

Spark supports more operations than MapReduce and it does not require the programmer to adapt a problem into a series of map and reduce operations. You can define an arbitrary set of inputs, an arbitrary number of transformations, and an arbitrary number of actions to produce results. There is no need for useless stages that might do nothing like one has to do when applying MapReduce iteratively.

Spark is fault tolerant. Dataset partitions can be regenerated if needed. The framework tracks what transformation was used to create the needed dataset and the input the transformation required. With that information, it can work backwards and recreate any missing RDD.

Finally, Spark provides a rich ecosystem and can support continuous data streams through Spark Streaming.

## References

- Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauley, Michael J. Franklin, Scott Shenker, and Ion Stoica. 2012., Resilient Distributed Datasets: A Fault-Tolerant Abstraction for In-Memory Cluster Computing, In Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation (NSDI'12). USENIX Association. – the original paper on Spark
- Job Scheduling, Apache Spark Documentation.
- RDD Programming Guide, Apache Spark Documentation.
- Cluster Mode Overview, Spark Documentation.
- Paco Nathan, Intro to Apache Spark, ITAS Workshop, Databricks
- Hands-on Tour of Apache Spark in 5 Minutes. Hortonworks
- Running Spark Applications, Cloudera 5.5.x documentation
- Sandy Ryza, Apache Spark Resource Management and YARN App Models, Cloudera Engineering Blog, May 30, 2014.
- Srini Penchikala, Big Data Processing with Apache Spark – Part 1: Introduction, InfoQ, Jan 30, 2015
- , Lakshay Arora, November 5, 2020.

This is an update to a document originally created on November 25, 2015 and revised on November 26, 2017.

Last modified November 23, 2023.