Apache Spark
–
An introduction to the basics concepts and machine learning practices

-

Created by Dominique Paul
for Prof. Johannes Binswanger
(Swiss Institute for International Economics and Applied Economic Research at the University of St. Gallen)

Content

1. About this text

This text offers a very basic introduction of what Apache Spark is and what is used for. It familiarizes the reader with the most basic concepts and why they are relevant. By no means should it be used as a general guide or complete introduction to Spark. Spark and the underlying concepts are very sophisticated and a more extensive guide should be sought to gain a proper understanding of what it means to work with Spark.

2. Introduction

2.1 What is Spark?

Spark is an open-source cluster computing software used for querying, analysing and transforming very big data sets stored on several servers. It is a language used for data analysis used when data sources become too big to analyse with regular languages such as R or Python. Spark originated as project at UC Berkeley in 2009 and was later made an open source project. After its release, Spark developed a broad developer community and moved into the Apache Software Foundation, a charitable organization dedicated to providing software for the public good, where it is being further developed as the foundation's biggest project.

Spark has enjoyed great popularity in the last years for three principle reasons. First, it is very simple to use due to access with APIs in different programming languages including R, Python and Scala. Second, it is fast. Spark is designed to run in memory and on disk. Depending on the task to be performed it can be up to 100 times faster than Hadoop, its biggest competitor. Third, Spark provides great support. It offers many libraries for interacting with different data bases, streaming live data and also a library for machine learning with out of the box ready tools. A second version of Spark released in 2016 made implementations in Spark easier and also introduced the so-called tungsten engine which allowed for a performance increase of several degrees of magnitude.

2.2 But wait, why can't we just analyse all data with Python or R?

Most data analyses performed can be done so on a single laptop with programming languages such as Python or R. However, some datasets are so big that we cannot store them on one hard drive, but instead must save them on servers in the cloud. Data stored across multiple servers cannot be analysed as easy with Python or R. If we would try to analyse the data with a script following Python or R rules, our computer from which we would be running it would have to constantly send and receive data from the individual servers, making it very slow. Also, if a server were to be unresponsive for a short time or break down completely, the entire script would fail.

Spark solves this problem by transforming our code into specific operations which are run on each cloud data server independently and which reduce inter-server communication to a minimum. This makes working with datasets distributed across several servers much more efficient and faster. Furthermore, through a method called parallelization single data points are stored multiple times across servers making the system robust to server failure.

3. Spark

3.1 Architecture of a Spark Program

Spark follows the master-slave architecture. This means that we differentiate between two server types in our server cluster: master and slave servers. The master server does not carry any data but controls the workflow of the other servers. The master server computes the most efficient order of physical tasks, launches tasks on the other servers, coordinates the work flow, and gathers responses from the servers to return results to the user. The slave servers, also called worker servers, contain and process the data. They execute the individual tasks as directed by the master node and without any communication with the other worker servers. This is called parallel execution and is what makes spark programs so fast.

The user defines the work in a Spark application, also called the Driver, which in turn creates a spark session, or spark context, which establishes the connection between the application and the Master Server. The driver prepares the context and declares the operations on the data, the Master Server optimizes these steps into physical tasks which take maximal advantage of the distributed server architecture. The master then submits the tasks to the individual worker nodes and collects the results when done, which are then returned to the user by the Master.

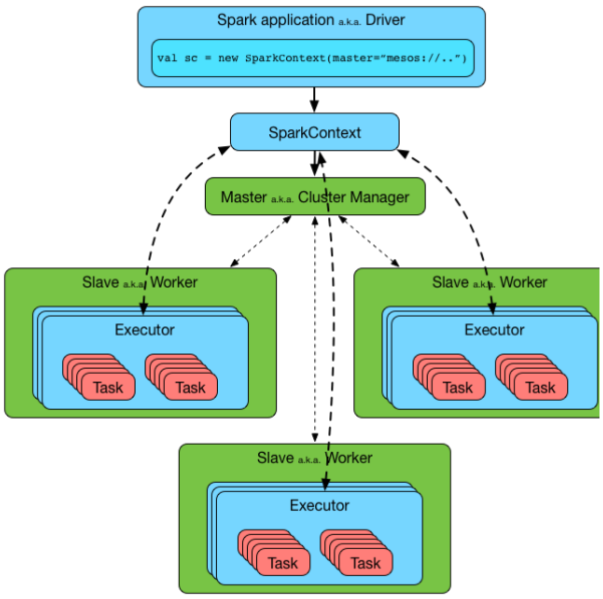


Fig. 1: The Structure of a Spark Program

(source: https://jaceklaskowski.gitbooks.io/mastering-apache-spark/spark-architecture.html)

3.2 Storing Files in a Spark Cluster

Before delving into the specifics of how Spark processes data, we will familiarize ourselves with some general principles of big data storage. We already saw that working with big data means working with data that must be stored across multiple servers as it is too big to be stored on one computer locally. This data must be kept in an organized manner which can be complicated with growing size. Several systems have been developed to tackle the associated problems, but we will focus on the Hadoop File Storage System (HDFS) which is very popular and often used for Spark clusters. It should be noted though, that Spark supports several file storage systems.

HDFS works by splitting a piece of data into many blocks and spreading these across the available servers. This allows for one piece of data to be processed in parallel, as multiple servers can process parts of the data stored on them. HDFS splits the original data into blocks of the same size which allows for all the blocks to be treated in the same way, and ensures that the processing time for each block is the same. Thereby, all servers are roughly equally fast and no bottlenecks are created where other servers wait for one server to finish a task. HDFS uses a block size of 128 megabytes, which has proven to be an ideal size, optimizing the trade-off between search and processing times for each block.

Mostly, servers in a Spark cluster are ordinary hardware and therefore anything but perfect. Individual blocks of data might be corrupted and single servers could break down due to errors. This, unfortunately, is the norm rather than the exception with hardware and has to be considered. To avoid the entire system's performance hinging on all servers working, HDFS stores each block several times across the different servers with a replication factor of three. By doing this, all relevant data can still be accessed if one server breaks down.

Using the analogy of a book can help understanding this data structure. To store the content of a book, HDFS would first break down the book into equal blocks, say chunks of 20 pages. These would be stored multiple times on the different slave servers (also called data nodes). The master server (or name node) would not store any pages of the book itself, but instead function as the table of contents, storing the metadata on which chunks of data exist and where to find them.

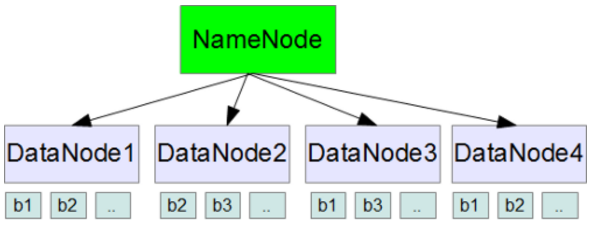


Fig. 2: Data storage with HDFS

(source: https://www.thegeekstuff.com/2012/01/hadoop-hdfs-mapreduce-intro/)

3.3 RDDs and MapReduce

Spark uses so-called Resilient Distributed Datasets (RDDs) which form the backbone of all data operations. RDDs are mostly not visible in Spark 2.0 as the user mostly interacts with other data structures built on top of RDDs, but their understanding is essential to follow how Spark programs work as they are still the fundamental building blocks.

Basically, a RDD is a collection of entities, similar to a table or a Dataframe which you might be familiar with from R or Python. A RDD contains rows of records such as strings, integers, floats etc. and is the prevalent form to store data in Spark. RDDs have three special characteristics.

First, RDDs are **partitioned** across several servers. This means that the information of a RDD is not necessarily stored together, but that one table is distributed across several servers. This partitioning is what allows for parallelization as multiple server can work through the data of one table, which might comprise several gigabytes or even terabytes. This is feature is hidden from the user, as to us the information of a RDD will always appear together if requested from the server.

Second, RDDs are **immutable**. This means that they cannot be changed. RDDs can only be created by one of two ways, by reading in new data or creating a RDD by transforming another RDD. The only option for the user to perform what we would see as changes to a RDD is to apply a transformation to an existing RDD with a new RDD being the output.

Third, RDDs are **resilient**. When a RDD is created through a transformation, it stores information on which RDDs were used to create it. This metadata is stored and creates what is called lineage, or a Directed Acyclic Graph (DAG). This means nothing else than that a flowchart is created, showing the process of how each RDD was constructed, going all the way back to the point where the data was initially read in. This means that if a server were to break down at some point of the work, the RDD can be reconstructed and that the program can keep running as usual and a failure can be averted. Furthermore, DAGs allow for higher processing efficiency through lazy evaluation, which is explained further below.

Since Spark 2.0 data is mostly stored and processed with DataFrames and Datasets instead of RDDs. Datasets are only available to the Scala and Java APIs, while DataFrames are available to all APIs. As the focus of this tutorial is on python we will focus exclusively on DataFrames. While RDDs are a collection of elements that can be operated on in parallel, DataFrames add to this by introducing a schema to the data making the storage easier for Spark. DataFrames are similar to DataFrames in R or Python, but with much stronger optimizations which run in the background and are hidden from the user.

Mappings and Reductions

We have now examined how data is stored with Spark in-program but not yet how we interact with the data. Despite differing partially, Spark works similar to the MapReduce programming model, which was first used at the beginning of the 2000s by Google in order to tackle large data sets which were emerging for the first time. MapReduce is very straightforward. As the name says, it breaks down all tasks into two types of operations: mappings and reductions.

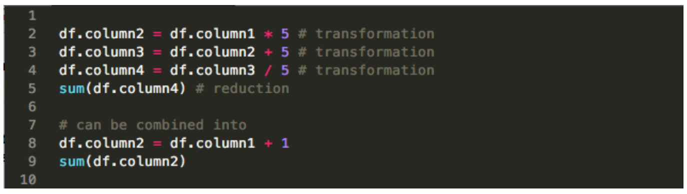
A mapping is a transformation applied to the existing data such as filtering or sorting. It can be applied individually on each server independently without exchanging data or information with other servers. Mappings are thus steps that can be performed in parallel and one of the reasons why Spark can process big data so fast.

Opposed to map methods are reduce methods. Reduce methods are summary operations collecting insights from the transformed data and returning them as an output to the user. This might be a calculation of say, the sum of numbers in a column. Reduce methods require communication between the servers and return actual results.

3.4. Lazy Evaluation

Typically, we will apply many transform methods on our data before applying any reduce methods. In Spark, these transform methods are never executed directly, but only saved in memory (remember that RDD transformations are saved in a Directed Acyclic Graph). Only when we execute a reduce method and demand an output from our program is it that these transformations are executed as well. This is called *lazy evaluation* and can generally be summarized under the rule that *data is not loaded until it is necessary*. As mentioned above, the Spark Driver searches for the most efficient method for the individual servers to perform the tasks. With lazy evaluation, this means that instead of evaluating each transformation as the driver comes across it in the code, it waits until a result is requested and evaluates the DAG of transformations to determine the most efficient method. By postponing the execution of the transformations until necessary the driver collects more information on what the user wants the application to do, the driver can find greater efficiencies.

Let's consider a very basic example where we apply some basic mathematical operations in a column and then request the sum of all values in the column (Fig. 3). If each operation were to have a computation cost of x, combining the first three tasks into one could reduce the total operations from four to two and speed up the process by a factor of two. Working with large sets of data, lazy evaluation can lead to large speed increases.



```

df.column2 = df.column1 * 5 # transformation
df.column3 = df.column2 + 5 # transformation
df.column4 = df.column3 / 5 # transformation
sum(df.column4) # reduction

# can be combined into
df.column2 = df.column1 + 1
sum(df.column2)

```

Fig. 3: Lazy Evaluation can combine three operations into one at execution time (own figure)

3. Machine Learning with Spark

Due to its Spark's nature of a big data tool a machine learning library was included from early on. However, the original MLlib library wasn't very scalable or extend able, so the new Spark ML library was introduced, inspired by Python's scikit learn. Unlike MLlib, Spark ML is compatible with the formerly introduced data frames, introduces much easier methods for hyperparameter tuning, and is also faster than MLlib by a large factor. Also, Spark ML introduced a new feature called pipelines to streamline machine learning processes, described in more detail further below. Currently the MLlib library remains much richer in functionality and is still commonly used to ensure backwards compatibility, but Spark ML is catching up fast and is on its way to becoming the predominant library for machine learning in Spark.

Spark's machine learning framework differs from only slightly from machine learning in other languages. These differences originate from the way Spark organizes data (that is via its RDDs and dataframes). Due to the properties of RDDs mentioned above which are also valid for dataframes (immutability & partitioning), we cannot simply write normal functions operating on and with dataframes. Instead, Spark uses so-called pipelines to chain operations together and create a workflow. These workflows can be depicted as Directed Acyclic Graphs (DAGs), which means nothing else than an order of actions following each other. Acyclic means that all elements of the graph flow in one direction and do not create repetitive loops, as otherwise the workflow could not end in a finite time.

The stages of a pipeline are either transformations or estimators. Transformations are similar to mappings described above. They are algorithms transforming one dataframe to another (e.g. taking a dataframes with word vectors and transforming them into a dataframe with a numerical representation). Estimators are algorithms that learn from a dataset and then create a transformer – a new algorithm that changes the data based on what the former algorithm learned. This new transformer created could then be applied to a dataframe containing features to create predictions. So, while transformers take dataframes as input and create a dataframe as an output, estimators take dataframes as input, but create a transformer as an output. You can picture estimators as the links of a pipeline which represent the machine learning model itself (e.g. a linear regression). They are usually called with a fit() method.

Pipelines chain these two types of operations together to a series to be performed on a specific kind of input data. There are typically two types involved. The first one, the *Pipeline*. is the training type where data is processed to then produce a model that is ready to use. The second type, the *PipelineModel* makes actual predictions. As an example, imagine that the first type is the process of getting your data into a ready format to apply a statistical model as well as training the parameters of the model. The produced PiplineModel also prepares the data inputted but then processes it through the transformer of the trained model to generate predictions.

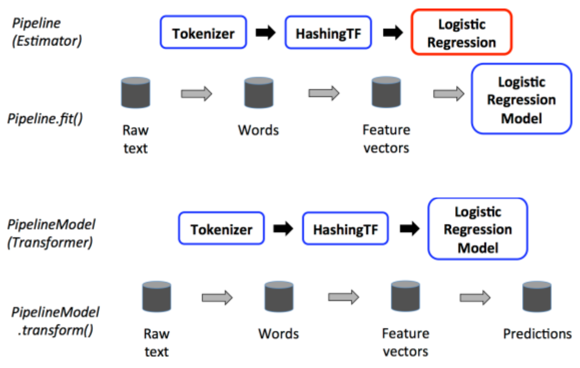


Fig. 4: An example of the two types of pipelines. Blue indicates transformations, red estimators

(source: https://spark.apache.org/docs/latest/ml-pipeline.html)

4. Differences to Hadoop MapReduce

Hadoop MapReduce is the prevalent big data tool for distributed storage and processing of data with simple commodity hardware. Released in 2006 it has since become part of the Apache software foundation and is open-source, just like Spark. Most companies that use big data sets and analytics work with Hadoop, effectively making it the industry standard.

Despite its dominant status, Hadoop has been challenged by Spark in recent years, due to the latter's great advantages in some areas, especially speed. While some[1] quote Hadoop installations at 50'000 opposed to 10'000 for Spark, the 2018 Stack Overflow developer survey[2] shows a much high popularity of Spark (66% v. 53.9%) while at the same time indicating that Hadoop developers remain more sought after (6.4% v. 4.8%). This section will point out the main differences between the frameworks.

Libraries and compatibility

[1] https://www.scnsoft.com/blog/spark-vs-hadoop-mapreduce
[2] https://insights.stackoverflow.com/survey/2018/#most-loved-dreaded-and-wanted

Many functions in Hadoop require the user to install a library first. This is cumbersome, but over time, an extensive set of libraries has accumulated allowing for a large amount of customizability. Spark on the other hand comes with in-built libraries for many important features such as machine learning, data streaming and graph analysis. Hadoop and Spark are mutually compatible. Spark can work with the Hadoop Filesystem as well as the general Hadoop ecosystem. It shares all of Hadoop's file compatibilities for data sources, file formats and business intelligence tools. Hadoop in turn can use Spark as well and even lists it as a module on its website. Pairing the two tools can provide for a very powerful tool for the big data applications.

Ease of Use

Spark can be used by developers of various backgrounds with the four language APIs it offers, despite being written in Scala. Through its REPL (Read-Eval-Print Loop) Spark also offers an interactive interface which makes it easier for newcomers to learn the language. Hadoop, on the other hand, is tricky to program and isn't as interactive as Spark. Tools like Pig and Hive can make learning Hadoop easier for beginners though.

Speed

This is where Spark shines. The key difference between Spark and Hadoop is that Hadoop reads from and writes to memory while Spark processes data in-memory. This causes Spark to work much faster, especially when there are several iterative cycles as it doesn't have to shuffle data around. Spark can sort 100TB of data 3x faster[3] than Hadoop and depending on the task can be more than 100x faster[4]. In particular, for Machine Learning applications Spark has exhibited some strong improvements in terms of speed[5].

Scalability

Both systems allow for massive amounts of data to be processed. Hadoop, however, is better suited for companies working with commodity hardware as its characteristic of writing to memory is better suited for these slower machines than Spark's in memory computation, This is because commodity servers have low working memory, which is required for Sparks in-memory operations, and as soon as this is exceeded Spark has to write to memory as well slowing it down. Big Data clusters tend to use Hadoop more frequently than Spark, even though no limit has been proven to Spark.

Cost

Both of the tools are open-source and thus free to use. Depending on whether servers are rented by an online service such as Amazon Web Services, Cloudera or Microsoft, both tools will incur costs. While differences exist depending on the choice of platform, Spark tends to be slightly more expensive as the servers rented must have sufficient RAM to process the data in-memory.

Conclusion

Spark emerges as the overall winner due to its ease of use, in-built libraries for many tasks and its much faster speed compared to Hadoop. Users must evaluate the needs of their project though. A project which does not rely on real-time data and where tasks may be run on servers overnight might not require the speed of Spark and should be considered to be run on Hadoop to exploit cost advantages.

[3] https://spark.apache.org/news/spark-wins-daytona-gray-sort-100tb-benchmark.html

[4] https://spark.apache.org/

[5]http://udspace.udel.edu/bitstream/handle/19716/17628/2015_LiuLu_MS.pdf?sequence=1

Hadoop MapReduce	Spark
• Written in Java	• Written in Scala
• Batch Processing	• Real-time processing
• Data stored on disk	• Stores data in-memory
• Rich library ecosystem	• Many built-in libraries
• Good for commodity hardware	• Up to 100x faster than Hadoop
	• Four programming APIs

Table 1: Comparing Spark and Hadoop

Principal sources and links for further readings:

https://www.manning.com/books/spark-in-action

https://www.ibmbigdatahub.com/blog/what-spark

https://mapr.com/blog/spark-101-what-it-what-it-does-and-why-it-matters/

https://databricks.com/product/getting-started-guide/dataframes

https://www.tutorialspoint.com/apache_spark/apache_spark_rdd.htm

https://spark.apache.org/docs/latest/rdd-programming-guide.html#transformations

https://www.scnsoft.com/blog/spark-vs-hadoop-mapreduce

https://www.edureka.co/blog/apache-spark-vs-hadoop-mapreduce