Introduction to Distributed Computing

By Jason Altieri

The rise of big data has required changes to the way that data is processed. Distributed computing is one approach used to meet expanding processing capacity needs driven by continually growing datasets. In *Designing Data Intensive Applications* Martin Kleppmann states, "Many applications today are **data-intensive**, as opposed to **compute-intensive**. Raw CPU power is rarely a limiting factor for these applications—bigger problems are usually the amount of data, the complexity of data, and the speed at which it is changing."[1] Where single machines or relational databases used to be sufficient for data processing and analysis, obtaining single machines that can handle today's quantity of data has become cost-prohibitive if not impossible. New software and processes have become necessary to allow efficient and effective use of this data.

In the actuarial world, these technologies have become important to support more advanced modelling on larger and more complicated data. The actuarial profession has been evolving to include more predictive analytics and these methods often require more compute power than traditional actuarial modelling. In addition, even the datasets used to support more traditional actuarial work have grown in recent years. For example, large reference datasets used for benchmarking and government databases, such as Medicare or Medicaid claims can be large. This combination of factors makes an understanding of these new technologies important for actuaries working in a modern data environment.

INTRODUCTION TO PARALLEL DISTRIBUTED COMPUTING

Distributed computing is a framework for handling large quantities of data and complex processes by increasing the amount of hardware applied to a task. Instead of using a single machine, a distributed computing system allows a network of machines to work together to complete a process. There are two ways to leverage the network to complete the process, which is referred to as parallelization. First, different nodes within the network can simultaneously work on different tasks in the process, as long as the tasks are not linearly dependent. Second, by taking a large task and splitting it up into smaller self-contained pieces, multiple nodes on the network can contribute to the completion of the same task. In practice, both of these approaches can be used within the same process provided there is a sufficiently large network.

Several factors have contributed to the rise of distributed computing, including increasingly large datasets, the popularity of statistical learning, and affordable access to hardware (often via cloud providers). Increasingly large datasets have made it more difficult to perform analysis on single computers, both because of memory and time constraints. Even if a single machine is capable of processing a large dataset it may be too slow, especially in industries that require the ability to react quickly to new data. Many common statistical learning algorithms applied to these large datasets also lend themselves to a distributed computing framework. Training these models, particularly on large datasets, can be memory and processing intensive. Additionally, some models benefit from running many iterations of the same model, and almost all statistical learning techniques utilize resampling to tune their accuracy. Both of these factors make statistical learning algorithms a perfect fit for parallelization. Finally, distributed computing has benefited from easier access to hardware. The rise of cloud computing resources has made large amounts of machine time and power broadly available to both companies and individuals. Cloud computing has also enabled users to access a large network of machines for just a limited amount of time and only pay for what they use. These factors, among others, have helped make distributed computing a popular tool for people who work with large quantities of data.

BENEFITS AND DRAWBACKS OF A DISTRIBUTED SYSTEM

Like with most technologies, there are both benefits and drawbacks to the use of distributed computing. Some of the advantages are:

- Distributed computing scales up effectively to very large datasets,
- acquiring large amounts of memory or processing power may be more affordable by networking a series of less expensive machines than buying one sufficiently powerful machine,
- externally maintained infrastructure such as cloud computing platforms can be leveraged and
- can decrease processing time, especially in non-linear pipelines.

The cost savings can be significant, particularly for a large cluster. According to *Designing Data Intensive Applications* "…cost is super-linear: a machine with twice as many CPUs, twice as much RAM and disk typically costs significantly more than twice as much."[2] However, there are disadvantages that can make distributed computing impractical to implement:

- Not all algorithms are good candidates for parallelization,
- lack of efficient scaling down to smaller data,
- overhead in getting programs up and running on additional nodes, and
- overhead in orchestrating the parallelization.

These disadvantages can be a significant barrier to the use of distributed computing in some cases. First, there are some algorithms and use cases that do not fit well in a distributed framework. The distributed computing framework requires the ability to separate data or distinct tasks, and this process does not work well in certain cases. Second, while a distributed approach scales up to very large data effectively, downscaling can be problematic. Each node on the network needs to be notified a task needs to be done, load up the environment to perform the task, and communicate results back to the main process. When the data is small this cycle can take more time than it would take to complete the process on a single machine. Additionally, there is a substantial amount of overhead involved in maintaining the parallelization. Systems need to exist to communicate what work needs to be done, manage the status of the processes on different nodes, and coordinate the compiling of results. Building and implementing a system capable of doing this is a significant investment, which can become prohibitive if a real need does not exist. Fortunately, systems exist to handle this communication and facilitate the use of distributed computing.

DISTRIBUTED COMPUTING TECHNOLOGIES

Several distributed computing technologies exist to help solve the problems related to managing a distributed computing system. There are two categories of solutions: MapReduce implementations and workflow managers. MapReduce implementations use a two-step process to break the data up and distribute it to different nodes, then re-aggregate it to determine a result. Workflow managers use dependencies between tasks to determine if there are tasks that are independent of the results of other tasks. The workflow manager then distributes the independent tasks to different nodes to allow for parallel completion of the tasks. The following is a non-exhaustive list of these solutions:

MAPREDUCE IMPLEMENTATIONS

- Apache Spark,
- Hadoop MapReduce,
- Disco,
- Dask and
- Teradata.

WORKFLOW MANAGERS

- Luigi,
- Airflow,
- Azkaban and
- Oozie.

The remainder of this article will focus on MapReduce, and dive specifically into Apache Spark.

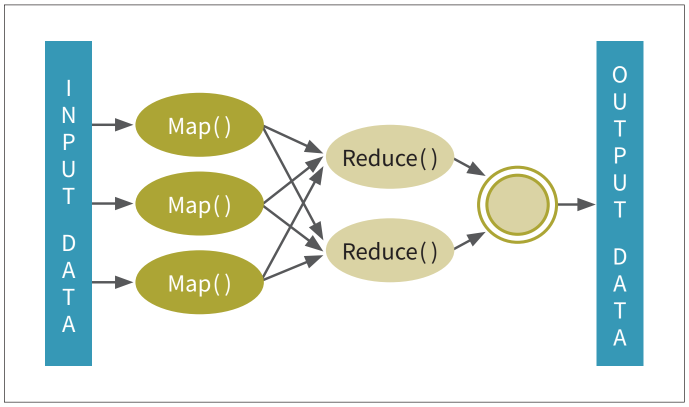
WHAT IS MAPREDUCE?

MapReduce is a process designed to facilitate parallel operations (See Figure 1). The original public implementation was part of the Hadoop ecosystem; however, the same general MapReduce concepts are used in other frameworks. As the name implies, the process is composed of two functions: A map function and a reduce function.

The map function breaks the data up into independent partitions. It then distributes these partitions to various nodes on the network for parallel processing. Each partition will output a group of key-value pairs that completes as much of the process as possible at the independent partition level. As each partition finishes processing, these key-value pairs need to be re-aggregated. This is the purpose of the reduce function. The reduce function pulls the results of each partition back into the main process and further aggregates them to determine the result at the full dataset level.

Manually implementing MapReduce is possible; however, it can be very difficult to do for even moderately complex processes. In order for a MapReduce process to be efficient, it is important to distribute the data to maximize the amount of work performed at the partition level. Additionally, it is important to minimize

Figure 1
The MapReduce Process



the frequency of re-aggregating the results, as that step does not benefit from the parallel framework. Fortunately, MapReduce solutions such as the ones listed above automatically perform this optimization using query planning and analysis tools.

OVERVIEW OF APACHE SPARK

There are many implementations of MapReduce-based distributed computing frameworks with different benefits and drawbacks. Depending on the infrastructure in place and the use case, the best implementation may vary. Apache Spark is one such implementation used as a more detailed example implementation of a MapReduce framework.

According to *Learning Spark*, "Apache Spark is a cluster computing platform designed to be fast and general-purpose."[3] It started as a research project at UC Berkeley back in 2009 by a lab working with Hadoop MapReduce. The researchers identified interactive querying and iterative development as a weakness of the Hadoop implementation and sought to improve it. Early results were positive; showing speed improvements in the 10x–20x range, and the performance has since improved to 100x faster on in-memory jobs. Spark was open-sourced in 2010 and became part of the Apache Software Foundation in 2013. Per the Spark documentation, it scales up to petabytes of data and clusters as large as 8000 nodes in practice.

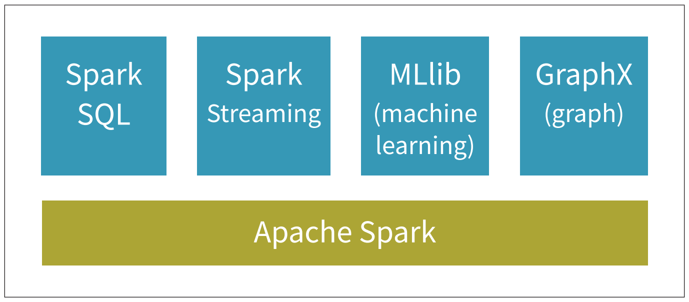
HOW DOES SPARK WORK?

On a technical level, Spark is written in Scala and runs on the Java Virtual Machine. It uses a concept called "Resiliently Distributed Datasets" (RDDs) to support its parallelized operations. According to *Learning Spark* "RDDs represent a collection of items distributed across many compute nodes that can be manipulated in parallel."[4]

Another key element of Spark is tightly integrated components. Spark is broken up into six main components:

- Spark Core: task scheduling, memory management, and other basic functions,

Figure 2
The Spark Ecosystem



- Spark SQL: querying and data manipulation for mostly structured data sources,
- Spark Streaming: API to work with live data updates,
- MLlib: scalable implementations of machine learning algorithms,
- GraphX: library for graph analysis and computation, and
- Standalone Scheduler: built-in cluster manager.

Spark Core is the foundational component that enables the system as a whole to function. From there, the other components act as extensions that allow the user to perform specific tasks such as querying or machine learning. The integration of these different components allows a user to switch between different types of tasks while remaining inside the Spark ecosystem.

Spark can also integrate with other common cluster managers such as Hadoop, YARN and Mesos. This allows Spark to be deployed inside of existing distributed computing infrastructure. Meanwhile the Standalone Scheduler allows deployment of Spark in cases where there is no existing distributed computing infrastructure.

BENEFITS OF SPARK

There are several benefits to using Spark as a distributed computing framework. For instance, knowledge of Scala programming is not necessary to work with data in Spark. There are convenient wrappers available to allow users to interact in a more familiar language such as python (PySpark) or R (SparklyR). Additionally, the APIs support the use of SQL syntax for data interaction. The availability of these common language interfaces helps reduce the learning curve for people looking to get started with Spark.

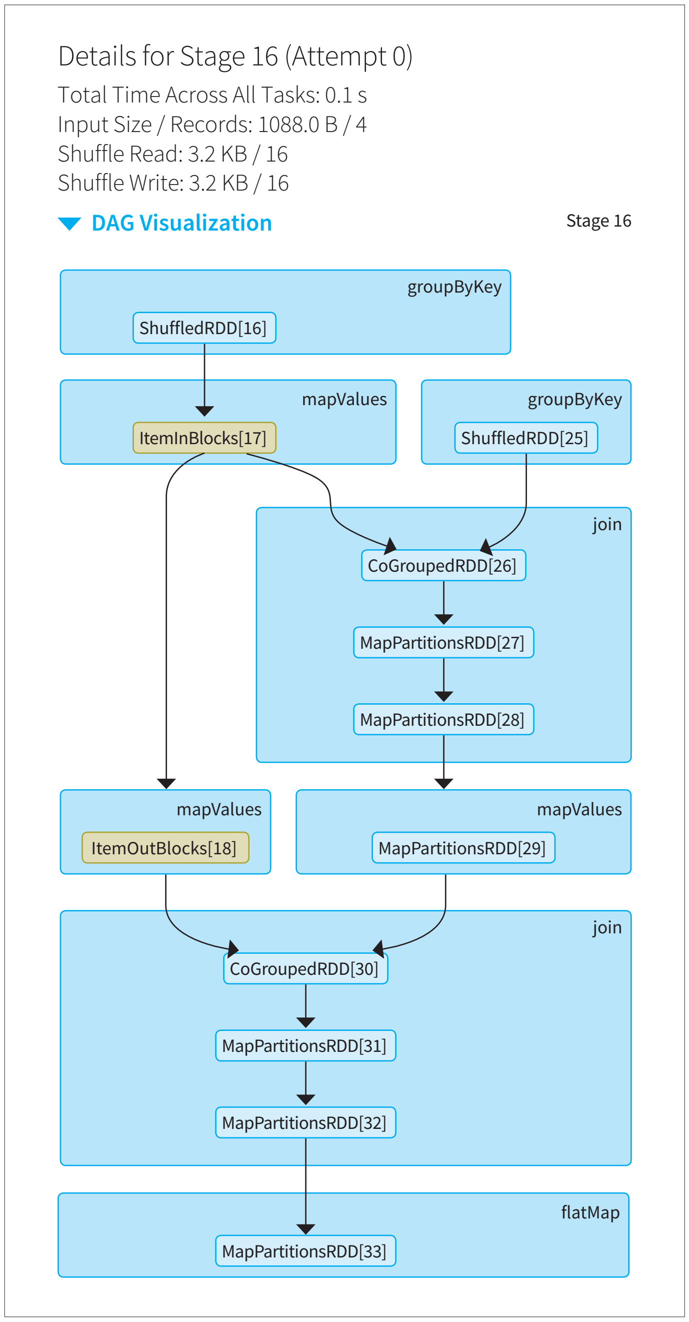
Spark also offers an interface that allows users to track the progress of jobs, data storage and query planning. The interface also offers a directed acyclic graph (DAG) to help visualize the execution of tasks. This allows for relatively straightforward performance monitoring and can assist with optimization of the system.

Additionally, while scaling down to smaller datasets can still be an issue; Spark handles data in the gigabyte range more effectively than some other options do. This makes Spark a viable choice for companies that have data in the gigabyte to terabyte range rather than the 100-terabyte range.

DRAWBACKS OF SPARK

Setting up the infrastructure necessary to run a Spark cluster can be challenging, especially on Windows-based systems. Spark was built to run on Linux, and its design choices reflect that. There is support for Windows; however, it is a clear second-class citizen and requires significantly more effort to implement and maintain.

Figure 3
Sample Directed Acyclic Graph (DAG)



Details for Stage 16 (Attempt 0)

Total Time Across All Tasks: 0.1 s
Input Size / Records: 1088.0 B / 4
Shuffle Read: 3.2 KB / 16
Shuffle Write: 3.2 KB / 16

In addition, while Spark scales down relatively effectively as a data manipulation and analysis language, the machine learning components do not. The performance of MLlib on smaller datasets does not compare favorably to common implementations in python and R. In particular, the Spark GBM implementation struggles on smaller datasets. As with any technology platform, it is important to understand the limitations of the specific implementation.

Finally, Spark lacks the flexibility of some lower-level frameworks, such as Dask, to build and control non-standard processes. Spark has some capabilities here, but they are mostly limited to the Scala language APIs.

GET STARTED

Getting started on working with Spark is easy. Databricks community edition offers free web-based notebooks running on top of pre-configured clusters in AWS. This removes the need to deal with setting up infrastructure for people who want to experiment with Spark. If you are interested in giving Spark a try, head over to *https://databricks.com/try-databricks*. ■

Jason Altieri, ASA, MAAA, is a data scientist with Milliman's PRM Analytics practice. He can be contacted at *Jason.Altieri@milliman.com.*

ENDNOTES

1 Kleppmann, Martin. 2017. *Designing Data-Intensive Applications: The big Ideas Behind Reliable, Scalable, and Maintainable Systems*. Sebastopol, Calif. O'Reilly & Associates, Inc. Pg. 3

2 Ibid. Pg. 146.

3 Karau, Holden, and Matei Zaharia, Andy Konwinski and Patrick Wendell. 2015. *Learning Spark: Lightning-Fast Data Analysis*. Sebastopol, Calif. O'Reilly & Associates, Inc.

4 Ibid. Pg. 3.

REFERENCES

Lockwood, Glenn K. "Map/Reduce Implementations." April 6, 2014. *users.sdsc.edu/~glockwood/comp/mapreduce.php*

Kempf, Rachel. "Members." Bizety. June 5, 2017. *www.bizety.com/2017/06/05/open-source-data-pipeline-luigi-vs-azkaban-vs-oozie-vs-airflow/*

"Apache Spark—Lightning-Fast Cluster Computing." Apache Spark—Lightning-Fast Cluster Computing, Apache, *spark.apache.org/*

"Comparison to Spark." Comparison to Spark—Dask 0.16.1 documentation, *dask.pydata.org/en/latest/spark.html*

"MapReduce." Hortonworks. *hortonworks.com/apache/mapreduce/#section_2*

"Analytics." What is MapReduce? IBM Analytics. *www.ibm.com/analytics/hadoop/mapreduce*